Reverse Level-Order Traversal

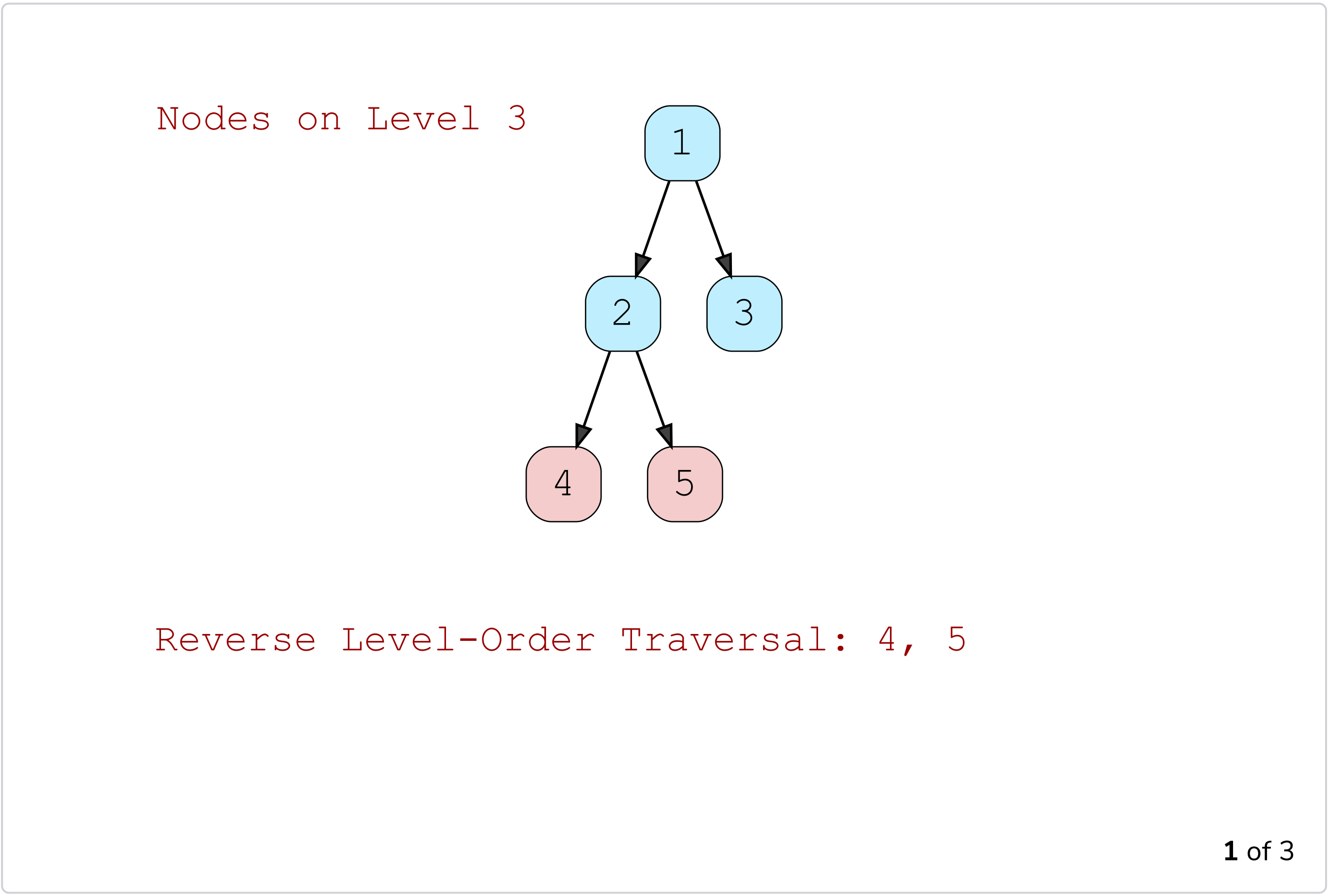
In this lesson, you will learn how to implement reverse level-order traversal of a binary tree in Python.

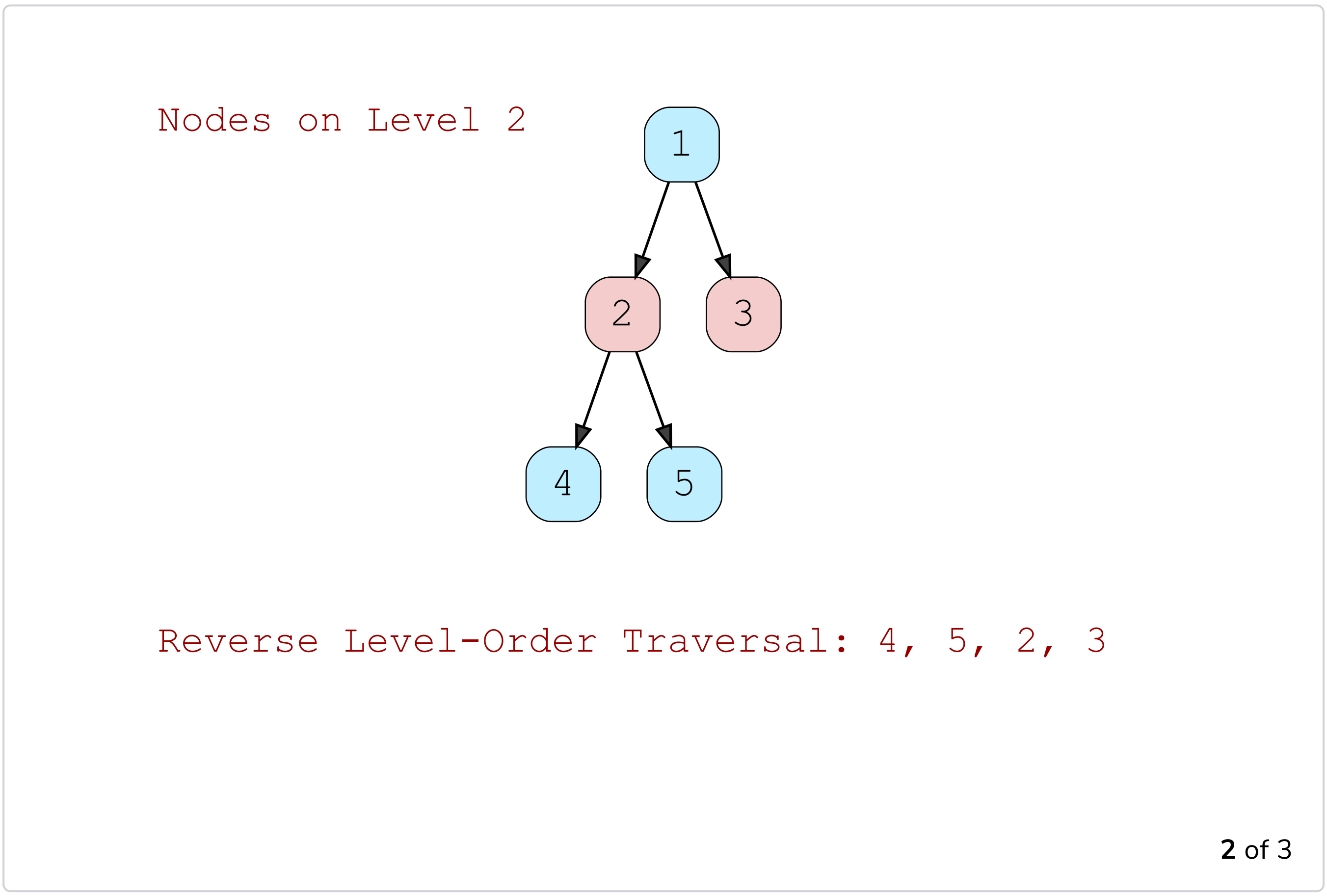
WE'LL COVER THE FOLLOWING

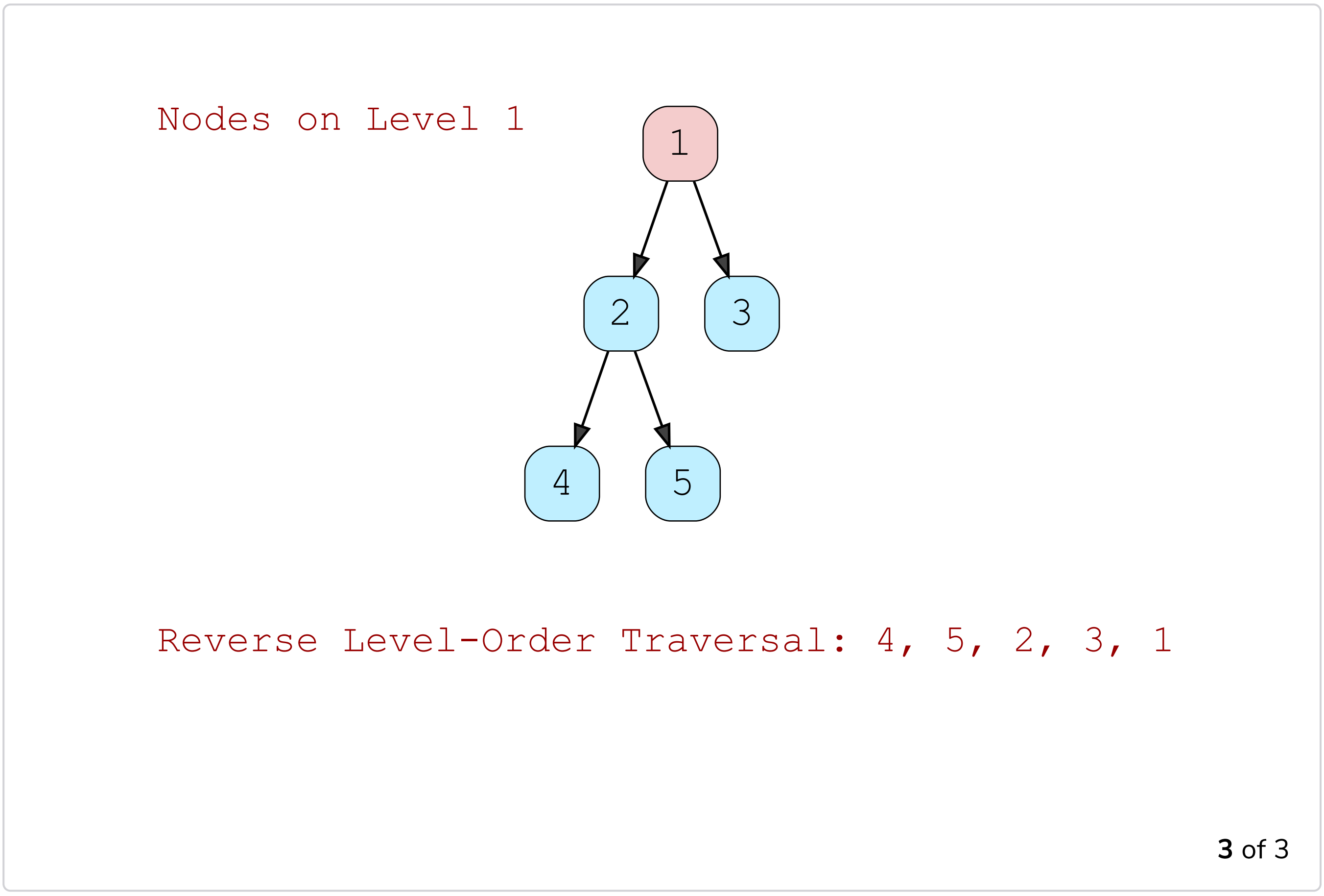
- Algorithm
- Implementation

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:

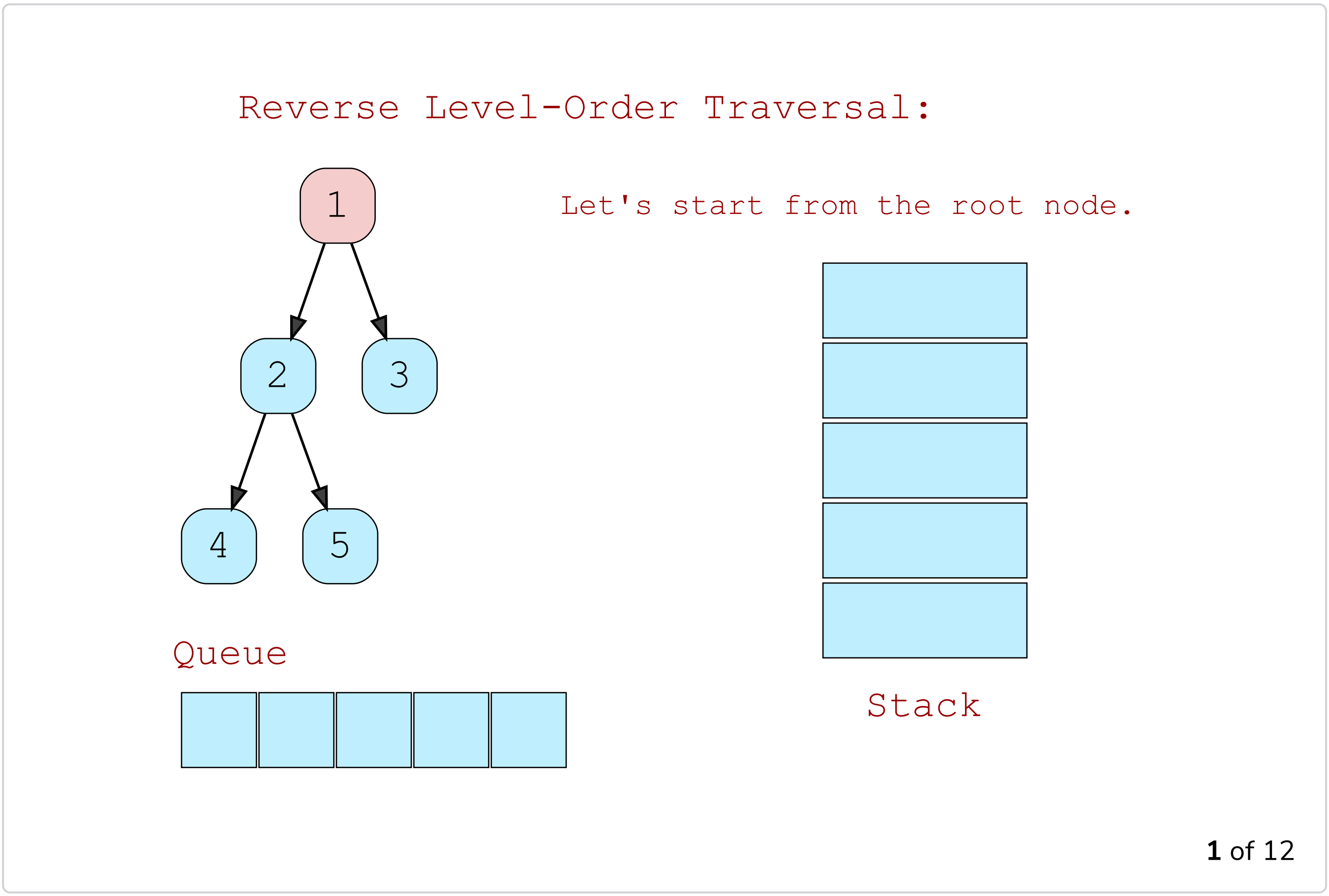



Nodes on Level 2
1
2
3
4
5
Reverse Level-Order Traversal: 4, 5, 2, 3
2 of 3


Nodes on Level 1
1
2
3
4
5
Reverse Level-Order Traversal: 4, 5, 2, 3, 1
3 of 3

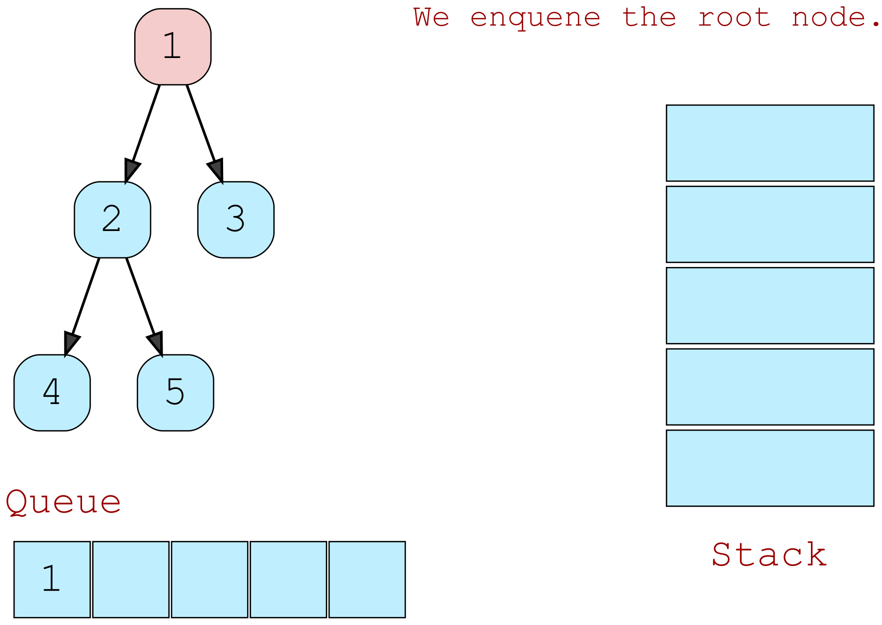
Algorithm

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:
1
2
3
4
5
Let's start from the root node.
Queue
Stack
1 of 12

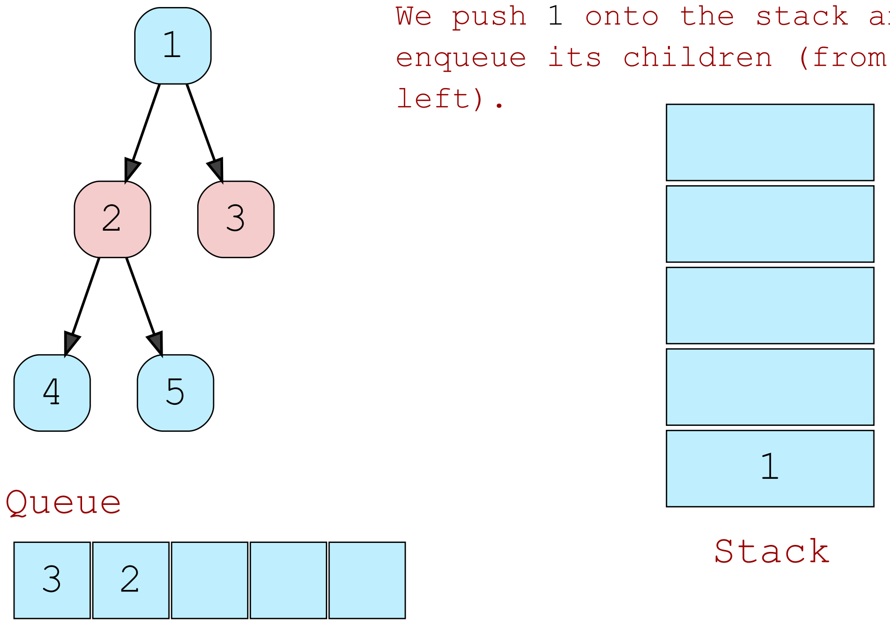
Reverse Level-Order Traversal:

We enquene the root node.



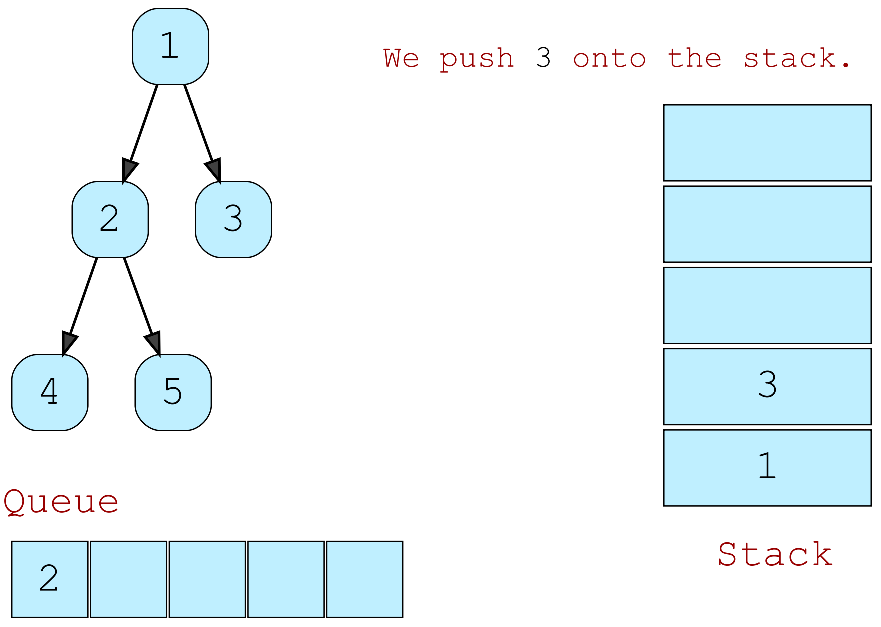
Reverse Level-Order Traversal:

We push 1 onto the stack and enqueue its children (from right to left).



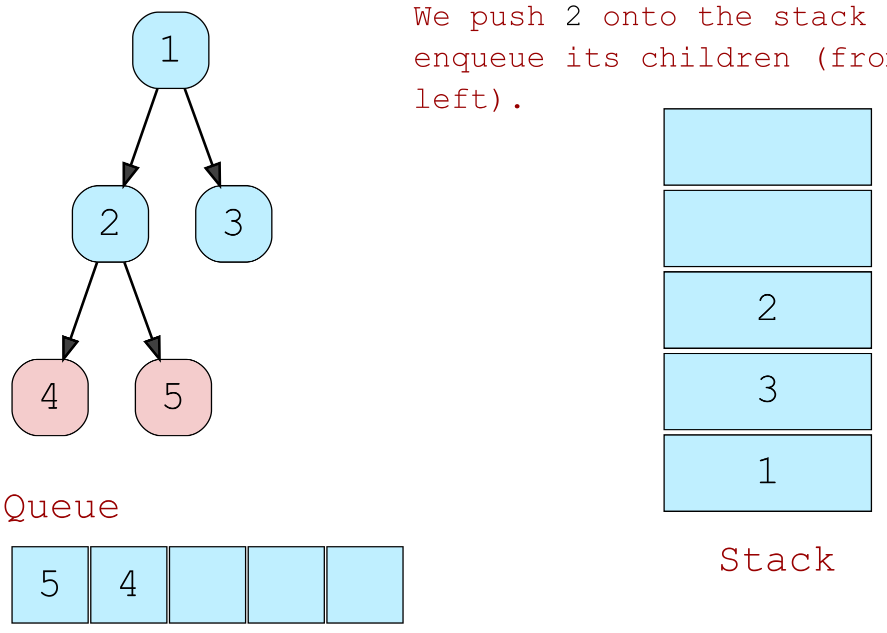
Reverse Level-Order Traversal:

We push 3 onto the stack.



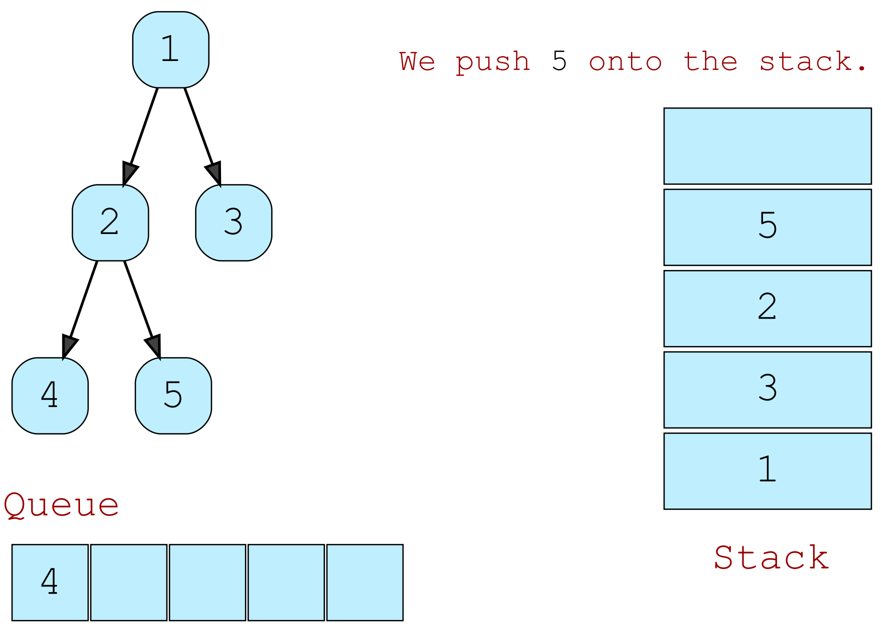
Reverse Level-Order Traversal:

We push 2 onto the stack and enqueue its children (from right to left).



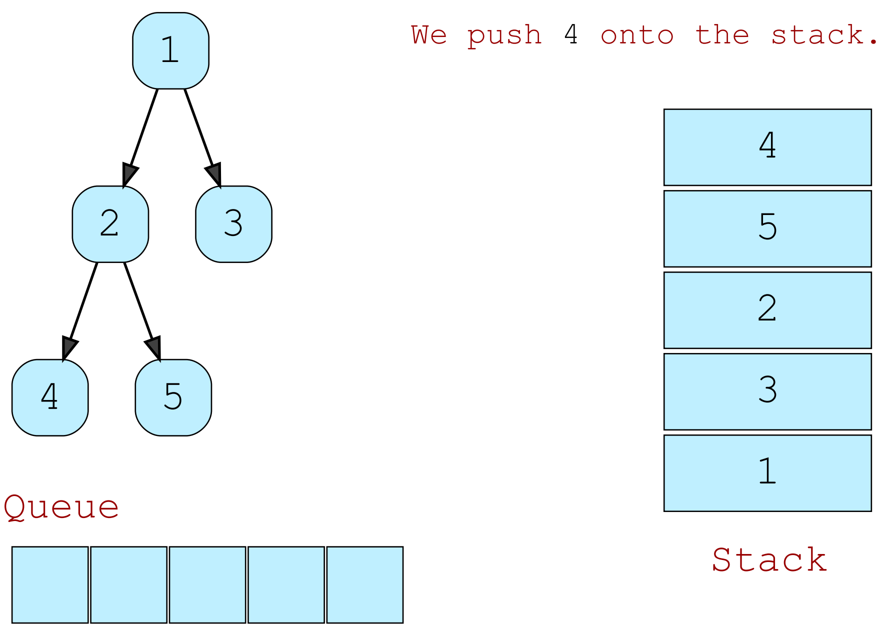
Reverse Level-Order Traversal:

We push 5 onto the stack.



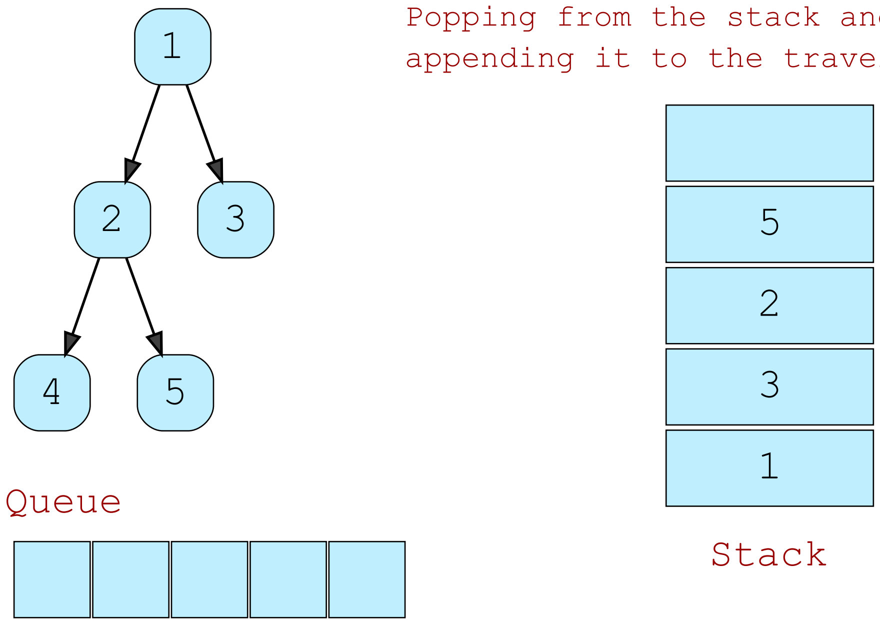
Reverse Level-Order Traversal:

We push 4 onto the stack.



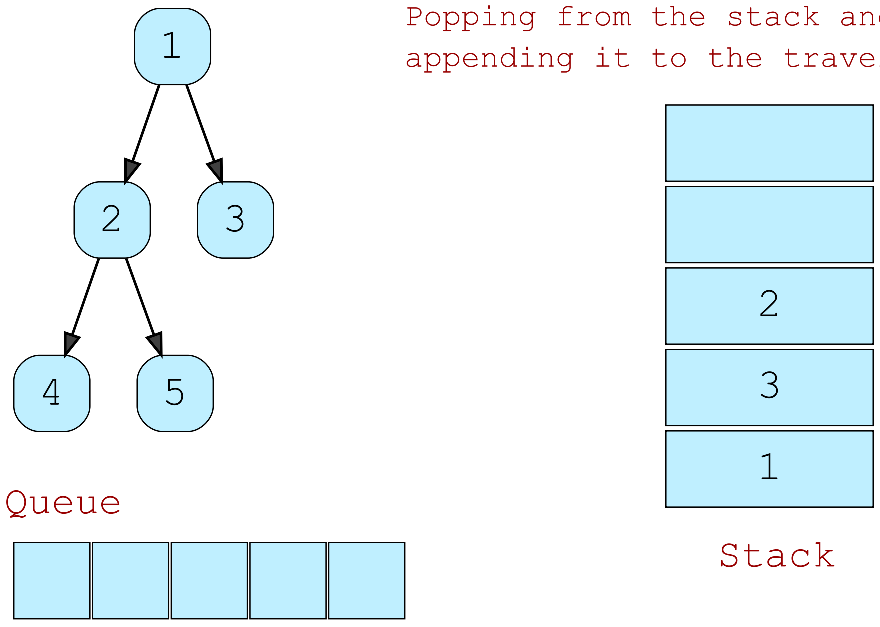
Reverse Level-Order Traversal: 4

Popping from the stack and appending it to the traversal

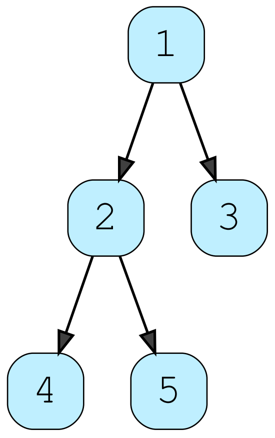


Reverse Level-Order Traversal: 4,5,

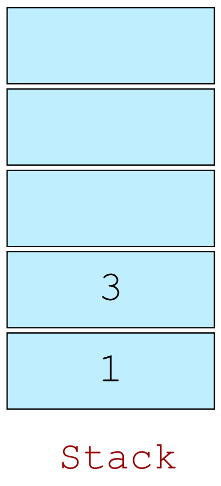
Popping from the stack and appending it to the traversal

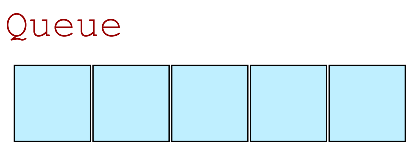


Reverse Level-Order Traversal: 4,5,2

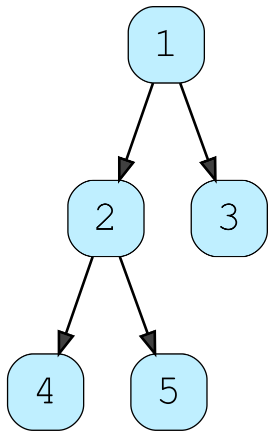


Popping from the stack and appending it to the traversal

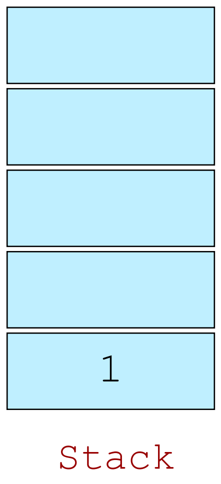


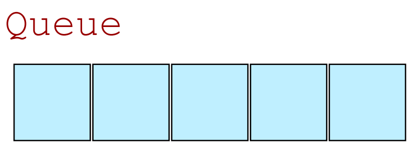


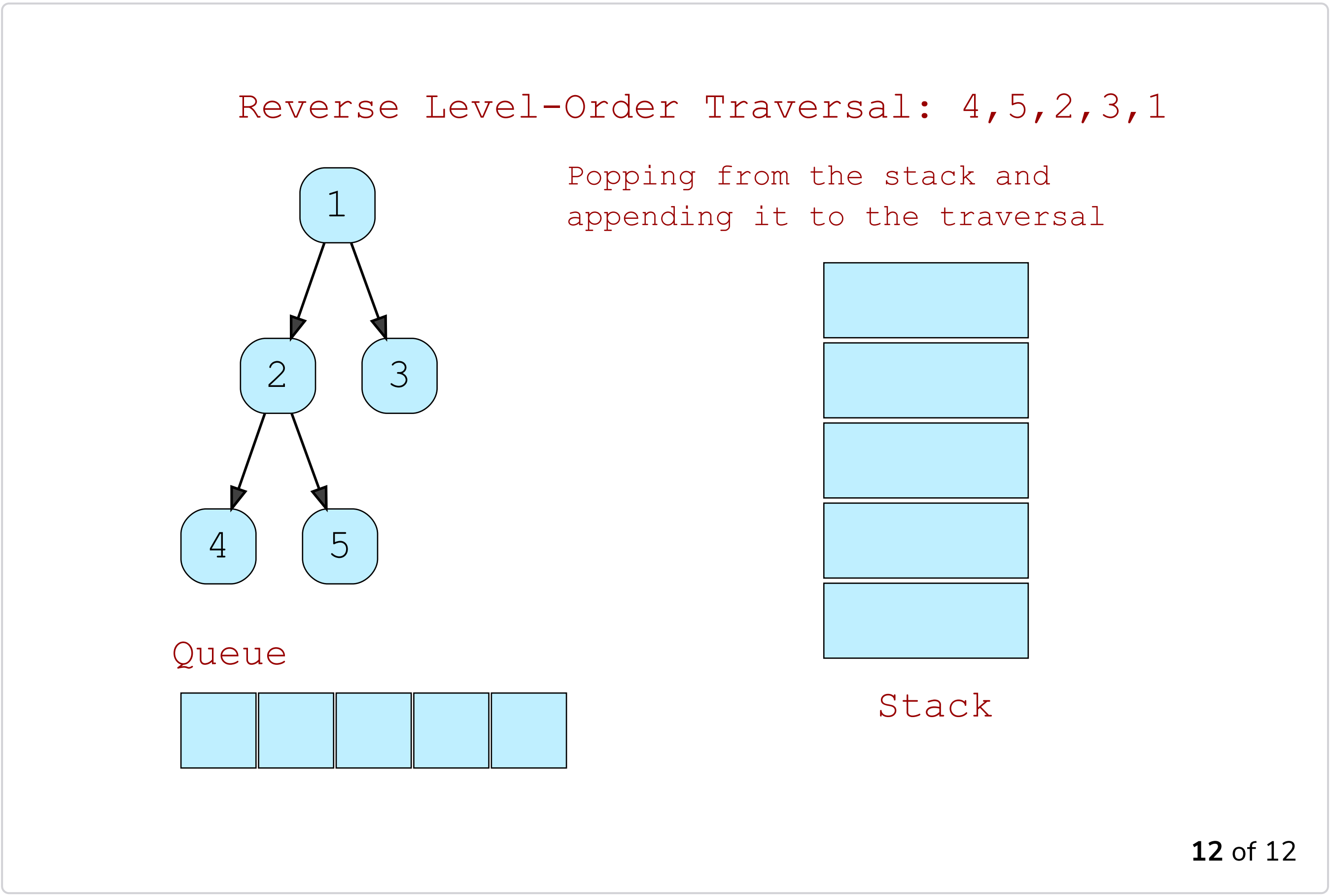
Reverse Level-Order Traversal: 4,5,2,3



Popping from the stack and appending it to the traversal






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:

```
class Stack(object):
    def __init__(self):
        self.items = []

    def __len__(self):
        return self.size()

    def size(self):
        return len(self.items)

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
    def is_empty(self):
        return len(self.items) == 0

    def __str__(self):
        s = ""
        for i in range(len(self.items)):
            s += str(self.items[i].value) + "-"
        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) + "-"

      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`.

```
class Stack(object):
    def __init__(self):
        self.items = []

    def __len__(self):
        return self.size()

    def size(self):
        return len(self.items)

    def push(self, item):
        self.items.append(item)

    def pop(self):
```

```python
        if not self.is_empty():
            return self.items.pop()


    def peek(self):
        if not self.is_empty():
            return self.items[-1]

    def is_empty(self):
        return len(self.items) == 0

    def __str__(self):
        s = ""
        for i in range(len(self.items)):
            s += str(self.items[i].value) + "-"
        return s

class Queue(object):
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop()

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        if not self.is_empty():
            return self.items[-1].value

    def __len__(self):
        return self.size()

    def size(self):
        return len(self.items)


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


class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == "preorder":
            return self.preorder_print(tree.root, "")
        elif traversal_type == "inorder":
            return self.inorder_print(tree.root, "")
        elif traversal_type == "postorder":
            return self.postorder_print(tree.root, "")
        elif traversal_type == "levelorder":
            return self.levelorder_print(tree.root)
        elif traversal_type == "reverse_levelorder":
            return self.reverse_levelorder_print(tree.root)

        else:
            print("Traversal type " + str(traversal_type) + " is not supported.")
            return False

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

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

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

    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) + "-"

        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"))
```

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