Level-Order Traversal

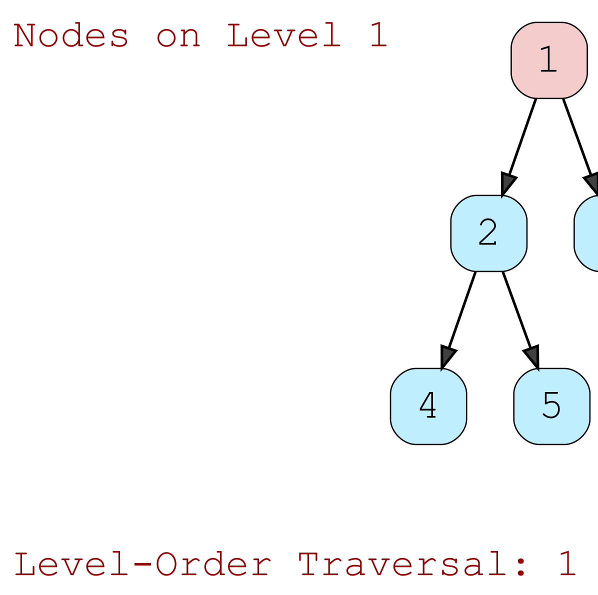
In this lesson, you will learn how to implement level-order traversal of a binary tree in Python.

WE'LL COVER THE FOLLOWING

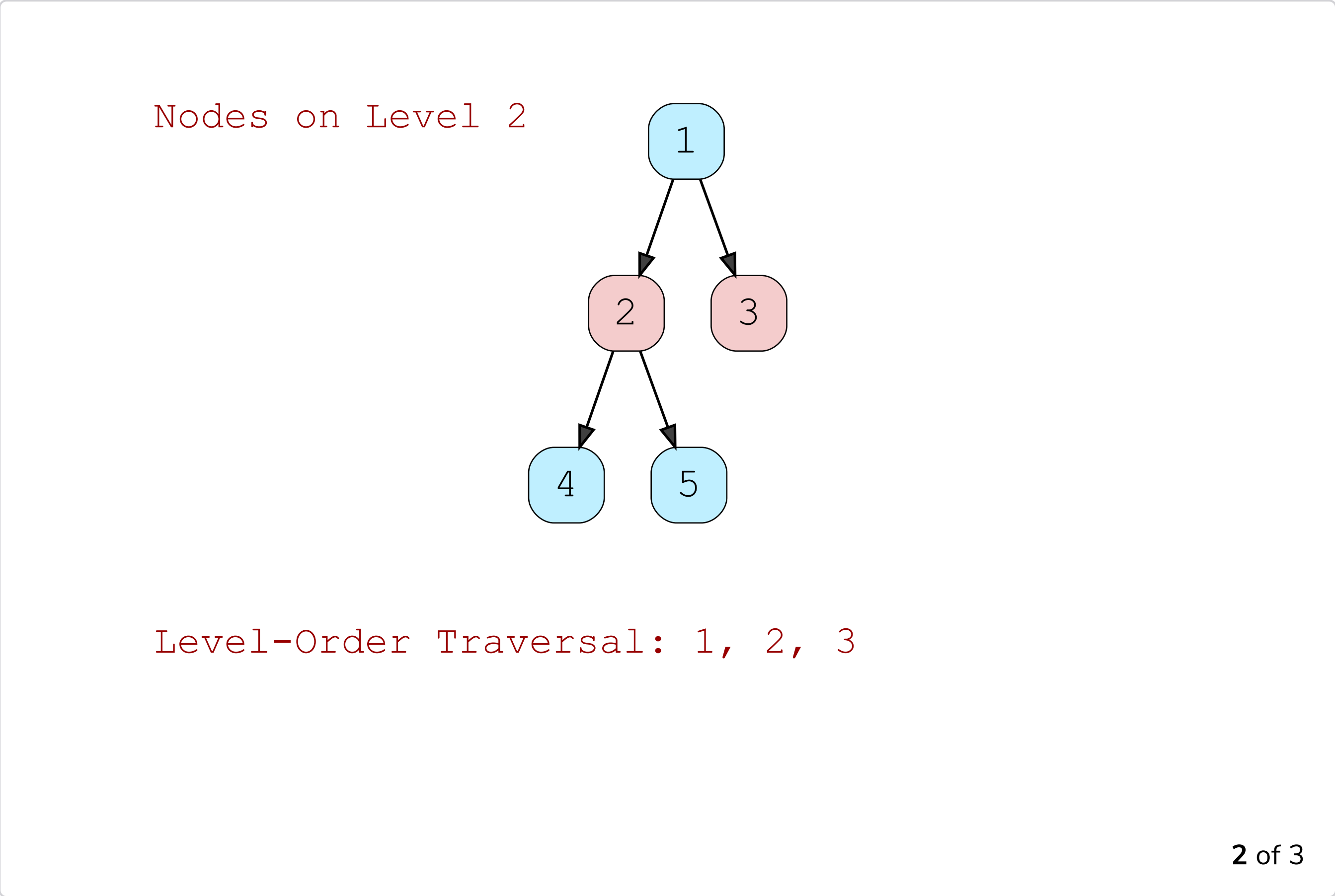
- Algorithm
- Implementation

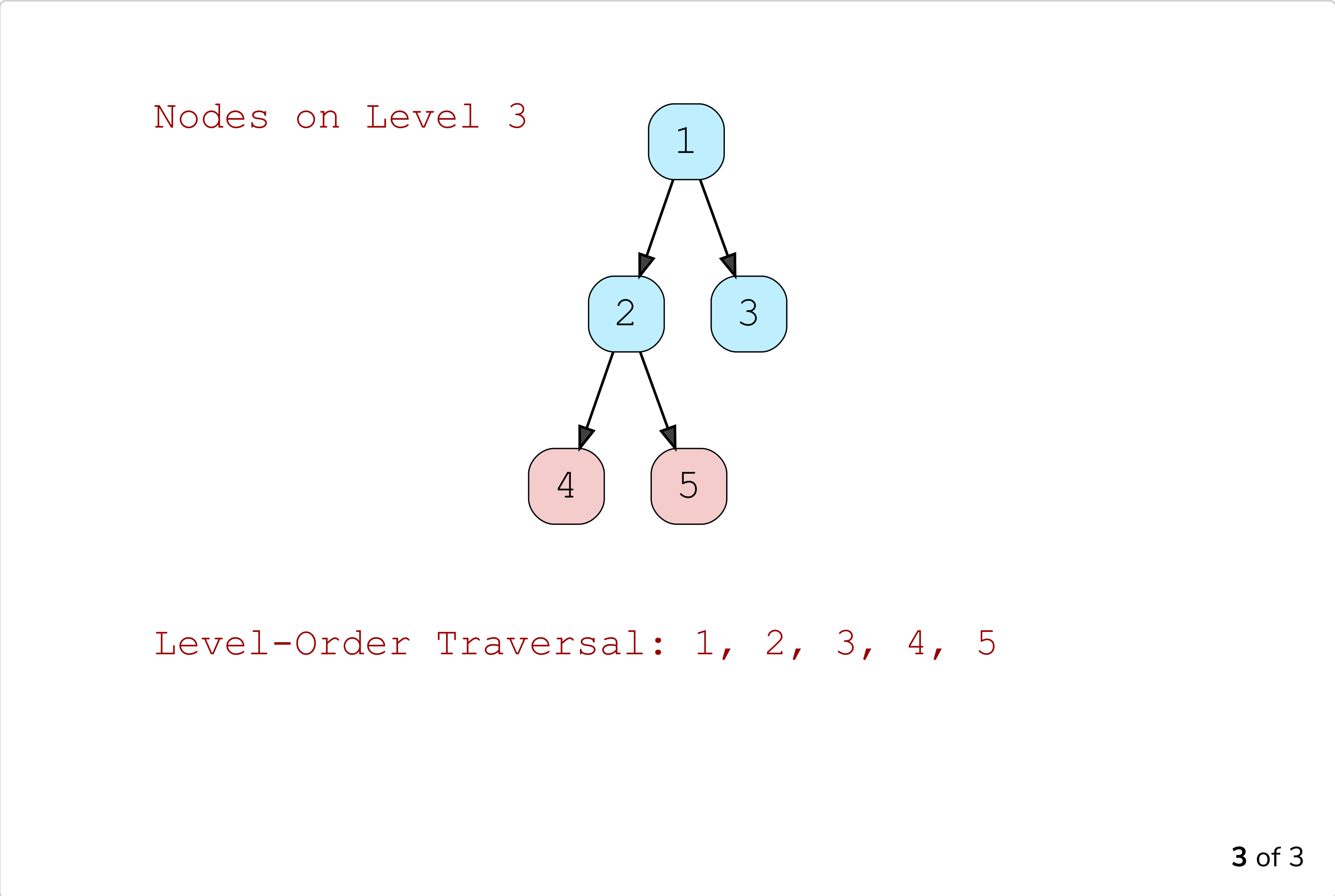
In this lesson, we go over how to perform a level-order traversal in a binary tree. We then code a solution in Python building upon our binary tree class.

Here is an example of a level-order traversal:



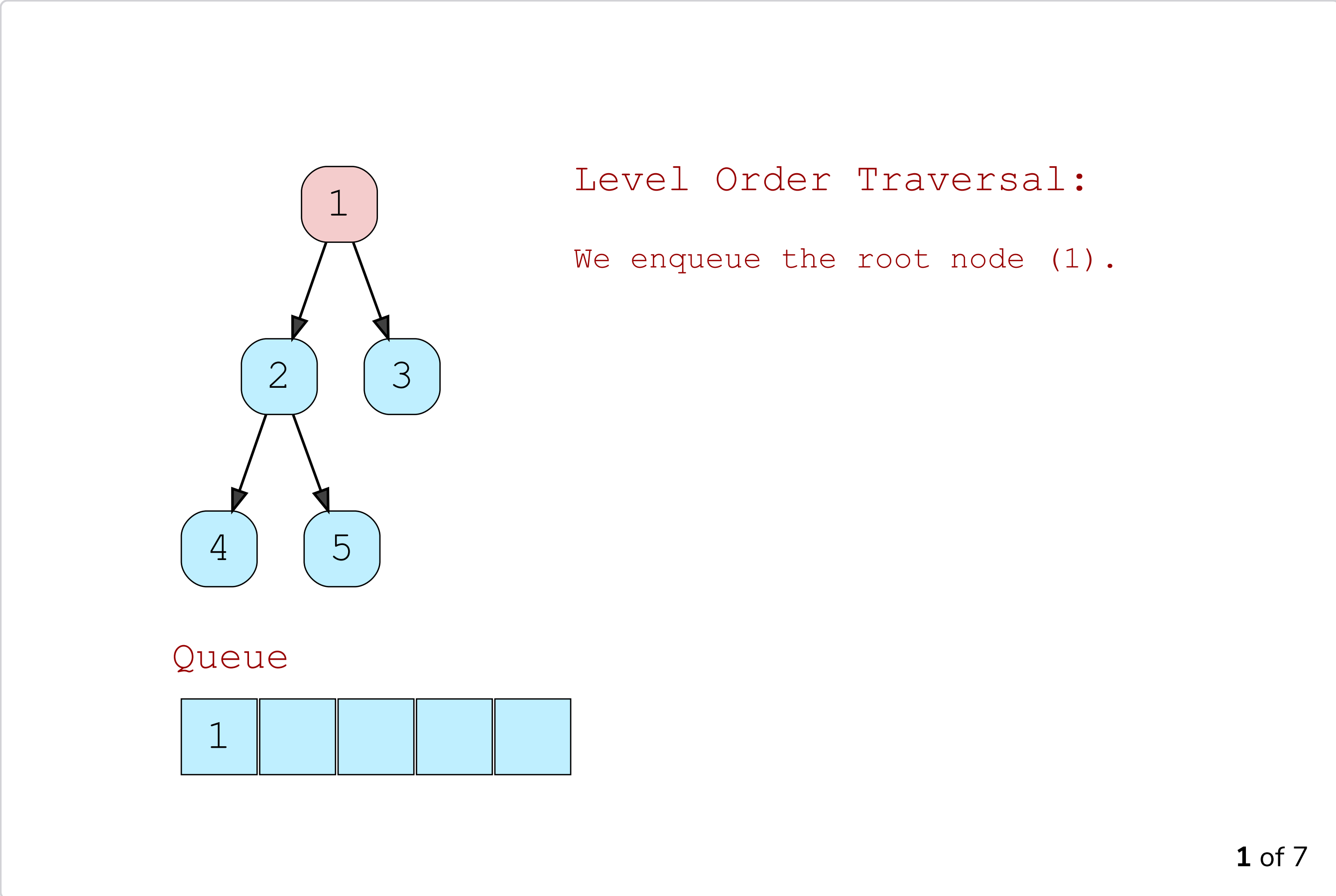
1 of 3

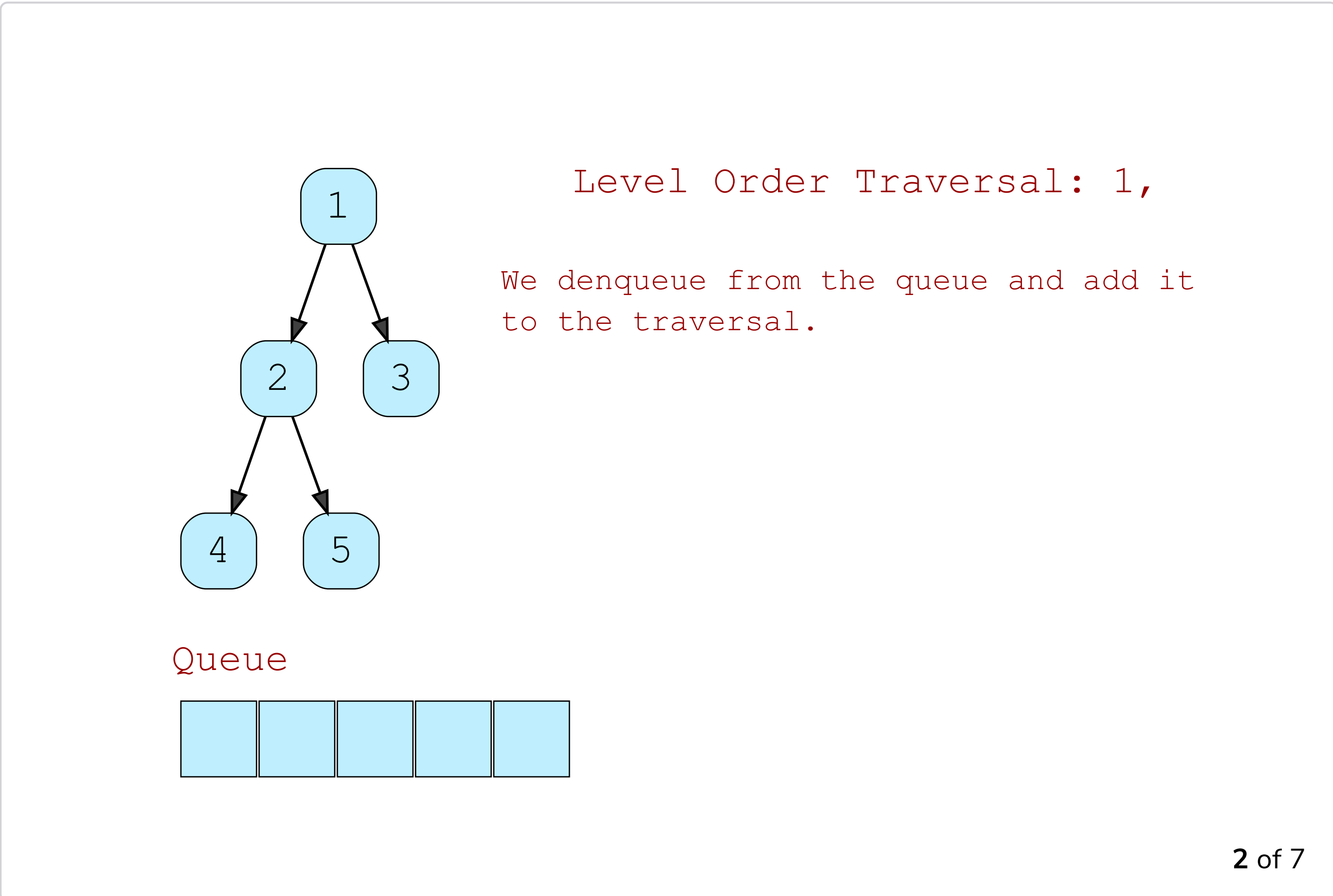

Nodes on Level 2
1
2
3
4
5
Level-Order Traversal: 1, 2, 3
2 of 3

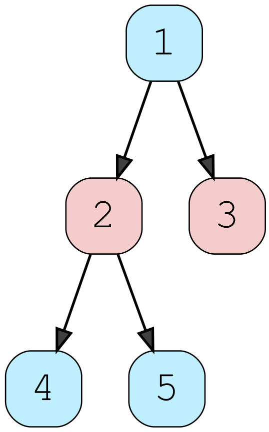

Nodes on Level 3
1
2
3
4
5
Level-Order Traversal: 1, 2, 3, 4, 5
3 of 3

Algorithm

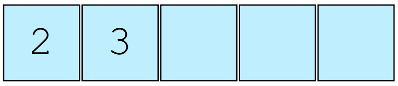
To do a level-order traversal of a binary tree, we require a queue. Have a look at the slides below for the algorithm:





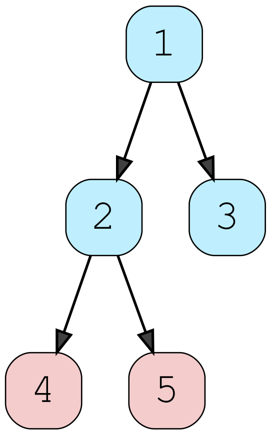


Queue

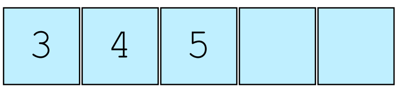


Level Order Traversal: 1,

We enqueue the children of the node we dequeued.

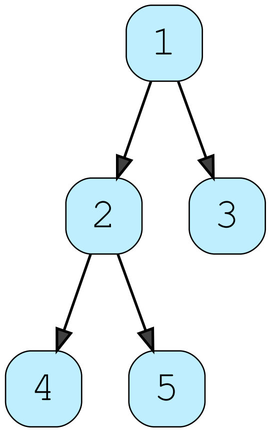


Queue

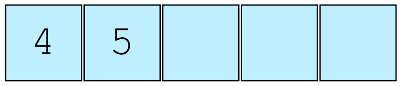


Level Order Traversal: 1,2

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

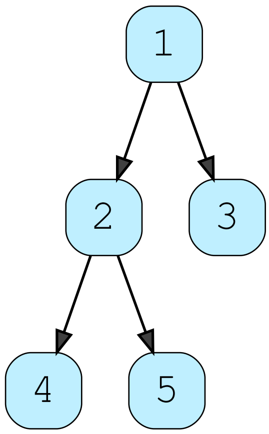


Queue

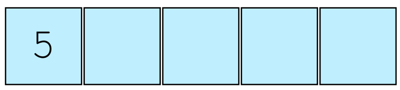


Level Order Traversal: 1,2,3

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

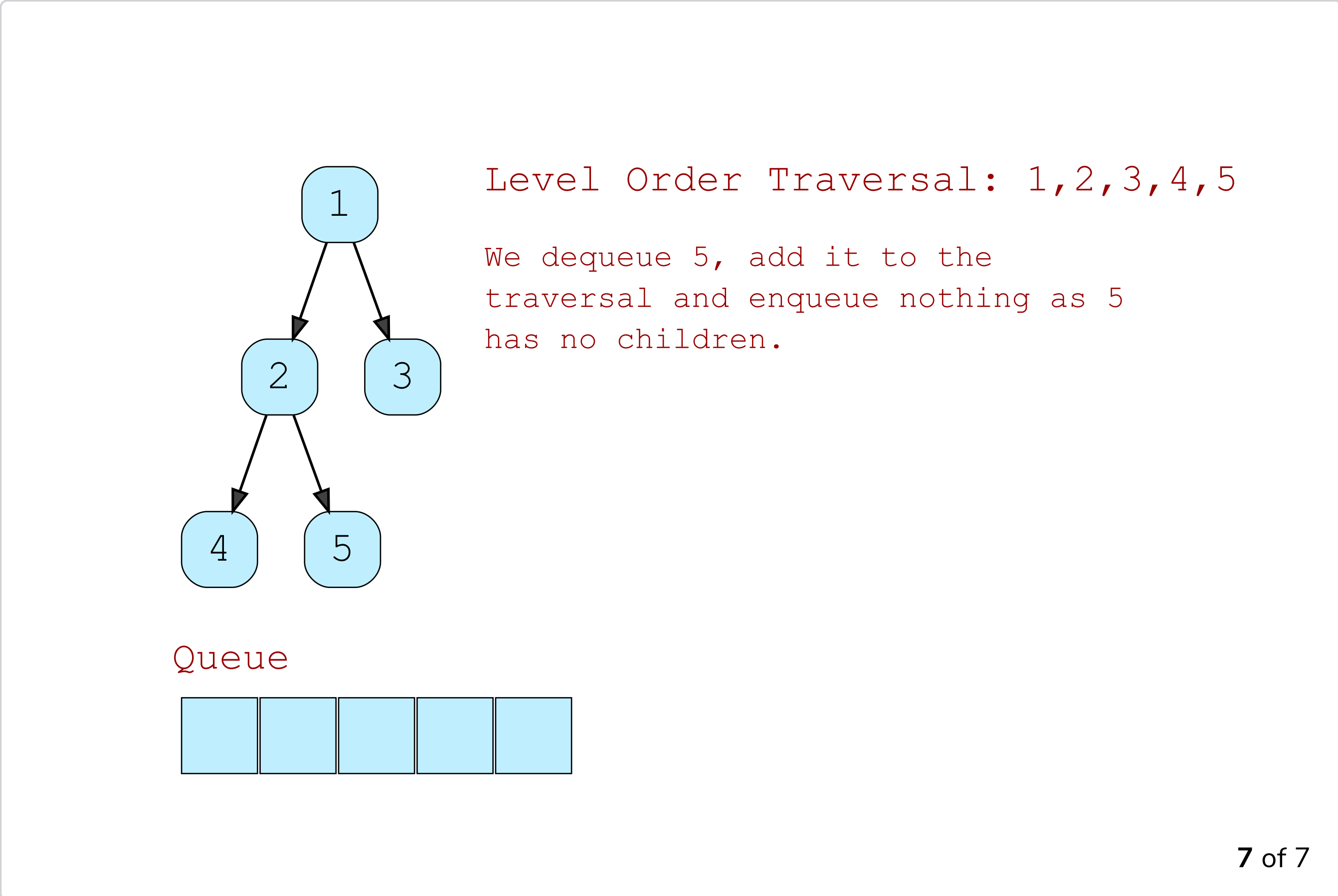


Queue



Level Order Traversal: 1,2,3,4

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



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.

```
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 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)
```

```
        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`.

```
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)

        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


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"))
```

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