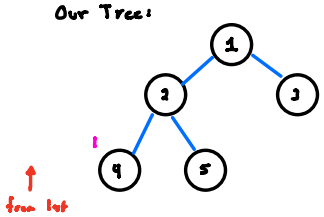
Reversal Level-Order Traversal

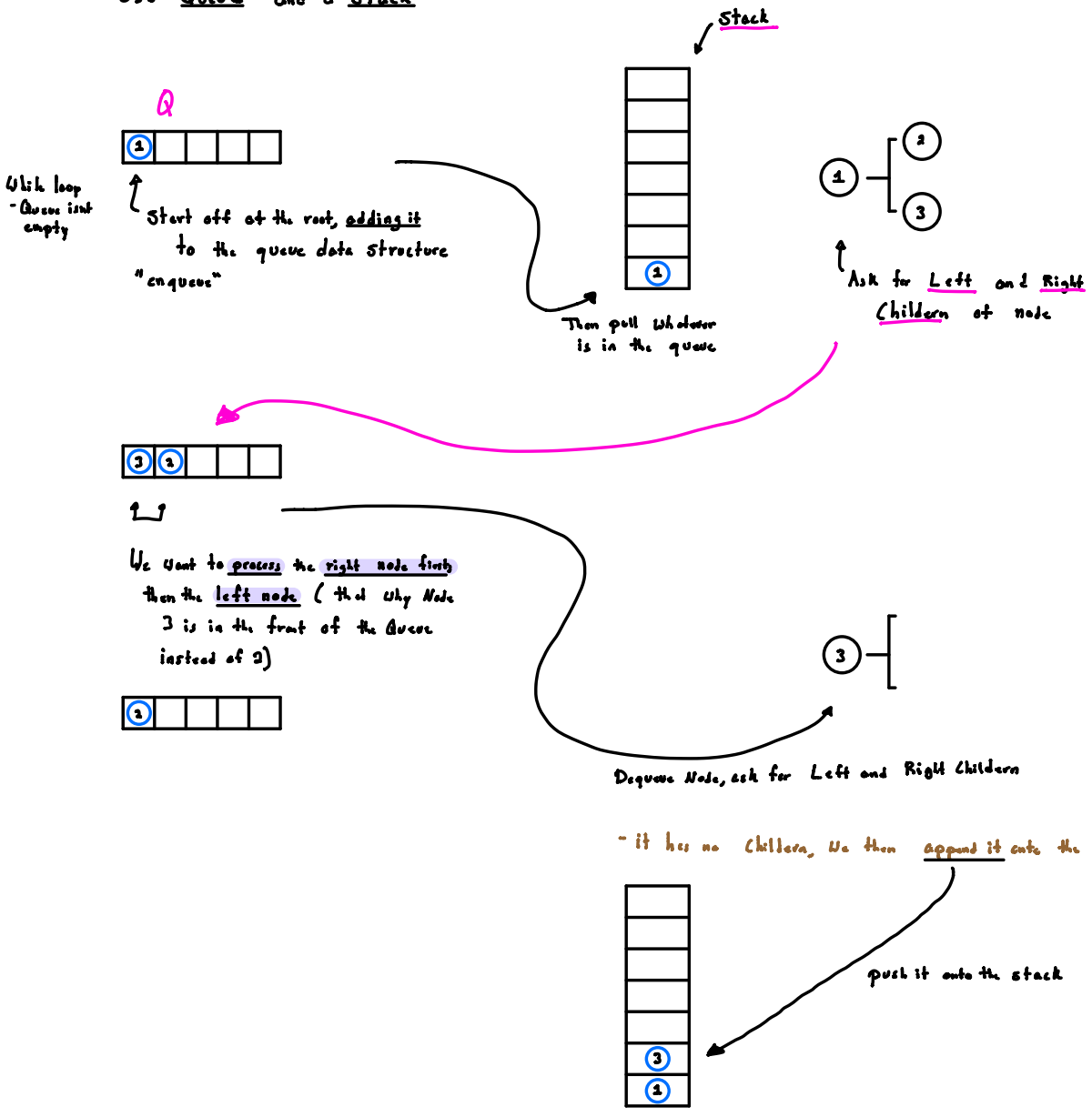
Reverse Level-Order Traversal: 4,5,2,3,1

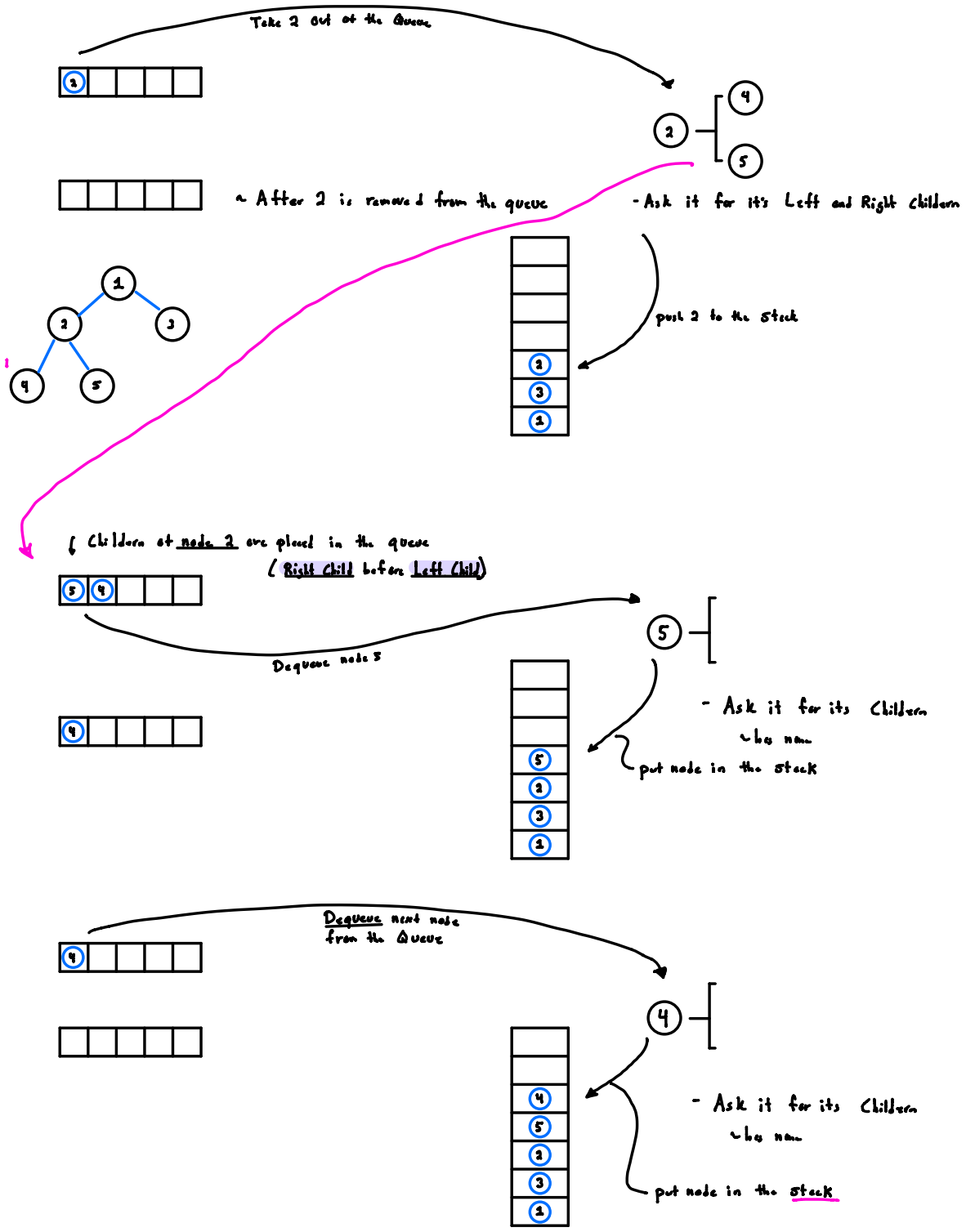
Our Tree:

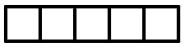


from last level up

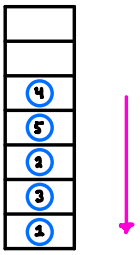
- Use Queue and a Stack




Take 2 out of the Queue
- After 2 is removed from the queue
- Ask it for it's Left and Right children
push 2 to the stack
(Children of node 2 are placed in the queue
(Right child before Left child)
Dequeue node 5
- Ask it for its Children
who now
put node in the stack
Dequeue next node
from the Queue
- Ask it for its Children
who now
put node in the stack



Here an empty Queue so While loop has ended



Then we will pop all the stack until its empty

reverse level-order Traversal

Code of Stack

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

    def __len__(self):   ~ Override method
        return self.size()

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

    def push(self, item):       ~ push onto the stack
        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]   ~ item at top of the stack.

    def is_empty(self):
        return len(self.items) == 0   ] if stack is empty

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

Function for reverse-order 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
```

Full Code:

```
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 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 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"))
```


4-5-2-3-1