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

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

    def pop(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]

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

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


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:
    def __init__(self, value=None):
        self.value = value
        self.right = None
        self.left = 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_print":
            return self.levelorder_print(tree.root)
        elif traversal_type == "reverse_levelorder_print":
            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 += (str(start.value) + "-")
            traversal = self.inorder_print(start.right, traversal)
        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

    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)

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

    def size(self):
        if self.root is None:
            return 0

        stack = Stack()
        stack.push(self.root)
        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

    class BST:
        def __init__(self):
            self.root = None

        def insert(self, data):
            if self.root is None:
                self.root = Node(data)
            else:
                self._insert(data, self.root)

        def _insert(self, data, cur_node):
            if data < cur_node.data:
                if cur_node.left is None:
                    cur_node.left = Node(data)
                else:
                    self._insert(data, cur_node.left)
            elif data > cur_node.data:
                if cur_node.right is None:
                    cur_node.right = Node(data)
                else:
                    self._insert(data, cur_node.right)
            else:
                print("Value is already present in tree.")

        def find(self, data):
            if self.root:
                is_found = self._find(data, self.root)
                if is_found:
                    return True
                return False
            else:
                return None
        def _find(self, data, cur_node):
            if data > cur_node.data and cur_node.right:
                return self._find(data, cur_node.right)
            elif data < cur_node.data and cur_node.left:
                return self._find(data, cur_node.left)
            if data == cur_node.data:
                return True


#                    1
#                   / \
#                  2   3
#                 / \  /\
#                4  5  6 7
#                         \
#                          8

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)
# tree.root.right.right.right = Node(8)
print(tree.print_tree("preorder"))
print(tree.print_tree("inorder"))
print(tree.print_tree("postorder"))
print(tree.print_tree("levelorder_print"))
print(tree.print_tree("reverse_levelorder_print"))
print(tree.height(tree.root))
print(tree.size_(tree.root))
```