Broken binary tree

* To calculate space complexity, look at the places where data structures are modified (added to/subtracted from)

Def brokenTree(node):

visited = []

# Visit O(n) nodes

find\_and\_eliminate\_in\_tree(node.left, visited)

find\_and\_eliminate\_in\_tree(node.right, visited)

def find\_and\_eliminate\_in\_tree(node, visited):

# check if node is in visited, if so, you know something is up

if node in visited: O(1)

# TODO something here

# Space complexity-wise, this is O(1)

visited.append(node)

Implement a function that takes two arguments: first an integer, block\_size, and second, an arbitrarily long string. It outputs a string where the string’s length is always a multiple of block\_size. It should do this by appending a "padding" to the end of the string. All characters in the padding should be the same. The character to use should be the length of the padding, e.g. "1", "22," "333," and so on. If the input string's length is already a multiple of block\_size, append a new block completely consisting of padding.

Example code:

Case 1

# we clarify if b\_s <= 0 raise an error

#case 2

# We take result = len(input) % b\_s

# output = absolute value of (b\_s - result) and then append concatenate output to input as string

=======================================Start Here ============================================

def padding(block\_size, input):  
if block\_size <= 0:  
 return "Block size can't be less than or equal to 0 but is {}".format(block\_size)  
 result = len(input) % block\_size  
 output = (block\_size - result)  
 for x in range(output):  
 input +=str(output)  
   return input  
  
# Main Method   
str1 = padding(5, 'World')  
print(str1)

========================================End Here =========================================

# Tests

# ========================

padding(4, “heyyheyyheyyheyy”) == “heyyheyyheyyheyy4444”

padding(4, “heyyheyyheyyhey”) == “heyyheyyheyyhey1”

padding(2, “hi”) == “hi22”

padding(3, "hi") == "hi1"

padding(2, ”Jelva”) == “jelva1”

assert len(output) % block\_size == 0

return output

padding(4, "hello") == "hello333"

padding(5, "world") == "world55555"

======================================= Jeff’s solution to PKCS#7

def padding(block\_size, input):

pad\_len = block\_size - (len(input) % block\_size)

return input + str([str(pad\_len)] \* pad\_len)

Implement a collection that supports two operations:   
1. Insert(value)  
2. GetMedian(): returns (int)   
GetMedian returns the median of all previously inserted values

c = Collection()

c.insert(5)

assert c.get\_median() == 5

c.insert(10)

assert c.get\_median() == 7.5

c.insert(15)

assert c.get\_median() == 10

c.insert(5)

assert c.get\_median() == 7.5

=================================Start here ===========================================

class StreamingMedian(object):

list = []

median = None

def insert(self, input):

#list.append(input)

#list.sort()

input = 3

list = [6]

low = 0

high = 1

mid = 0

low = 0

high = len(list)

mid = 0

while low < high and mid > low and mid < high:

mid = ((low + high) // 2) + low

if input < list[mid]:

high = mid

else:

low = mid

list.insert(mid, input)

if len(list) % 2 == 0:

first\_value = list[(len(list) / 2) - 1]

second\_value = list[len(list) / 2]

self.median = (first\_value + second\_value) / 2

else:

self.median = list[len(list) // 2]

def get\_median(self):

return self.median