```
# LAB 5
# REMINDER: The work in this assignment must be your own original work and must be
completed alone.

class Node:
    def __init__(self, value):
        self.value = value

     self.left = None
        self.right = None

    def __str__(self):
        return
("Node({})".format(self.value))

    __repr__ = __str__


class
BinarySearchTree:
    '''
        >>> my_tree = BinarySearchTree()

>>> my_tree.isEmpty()
        True
        >>> my_tree.insert(9)

>>> my_tree.insert(5)
        >>> my_tree.insert(14)
        >>>
my_tree.insert(4)
        >>> my_tree.insert(6)
        >>>
my_tree.insert(5.5)
        >>> my_tree.insert(7)
        >>>
my_tree.insert(25)
        >>> my_tree.insert(23)
        >>>
my_tree.getMin
        4
        >>> my_tree.getMax
        25

>>> 67 in my_tree
        False
        >>> 5.5 in my_tree
        True

      >>> my_tree.isEmpty()
        False
        >>>
my_tree.getHeight(my_tree.root)   # Height of the tree
        3
        >>>
my_tree.getHeight(my_tree.root.left.right)
        1
        >>>
my_tree.getHeight(my_tree.root.right)
        2
        >>>
my_tree.getHeight(my_tree.root.right.right)
        1
        >>>
my_tree.get_closest(18)
        14
        >>> my_tree.get_closest(19)

23
        >>> my_tree.get_closest(5)
        5
        >>>
my_tree.get_closest(72)
        25
```

```
        >>> my_tree.get_closest(7)

7
        >>> my_tree.get_closest(8)
        9
    '''
    def __init__(self):

      self.root = None


    def insert(self, value):
        if self.root is None:

     self.root=Node(value)
        else:
            self._insert(self.root, value)



def _insert(self, node, value):
        if(value<node.value):

if(node.left==None):
                node.left = Node(value)
            else:

  self._insert(node.left, value)
        else:
            if(node.right==None):

    node.right = Node(value)
            else:
                self._insert(node.right,
value)


    def isEmpty(self):
        # YOUR CODE STARTS HERE
        # check if the
root node of the tree is empty or not
        # return True if it is
        if self.root is
None:
            return True
        # return False if it is not
        else:

return False
        pass



    @property
    def getMin(self):
        # YOUR CODE
STARTS HERE
        # check if the tree is empty or not
        if self.isEmpty() is True:

          return None

        # if left node of root does not exist, return the root value
because it is the min value
        if self.root.left is None:
            return
self.root.value
        # else call the getMinHelper to further check the left side of the
tree
        else:
            return self.getMinHelper(self.root.left)
        pass


def getMinHelper(self, node):
```

```
        # return the current value if there is no left node

    if node.left is None:
            return node.value
        # else keep calling
getMinHelper to further check the left side of the tree
        else:
            return
self.getMinHelper(node.left)
        pass


    @property
    def getMax(self):
        #
YOUR CODE STARTS HERE
        # check if the tree is empty or not
        if self.isEmpty()
is True:
            return None

        # if right node of root does not exist, return the
root value because it is the max value
        if self.root.right is None:
            return
self.root.value
        # else call getMaxHelper to further check the right side of the tree

       else:
            return self.getMaxHelper(self.root.right)

        pass



def getMaxHelper(self, node):
        # return the current value if there is no right node

     if node.right is None:
            return node.value
        # else keep calling
getMaxHelper to further check the right side of the tree
        else:
            return
self.getMaxHelper(node.right)
        pass



    def __contains__(self, value):

# YOUR CODE STARTS HERE
        # if the value of the root node equals to the checked value,
return true
        if self.root.value == value:
            return True

        # if the
root nodes' left node exists
        if self.root.left is not None:
            # if the
checked value is less than the root node's value return the result of containsHelper for left
node
            if value < self.root.value:
                return
self.containsHelper(self.root.left, value)

        # if the root nodes' right node exists

      if self.root.right is not None:
            # if the checked value is more than the root
node's value return the result of containsHelper for right node
            if value >
self.root.value:
                return self.containsHelper(self.root.right, value)
```

```
# returns False if no "True" conditions are met
        return False

pass

    def containsHelper(self, node, value):
        # if the value of the current node
equals to the checked value, return true
        if node.value == value:
            return
True

        # if the current nodes' left node exists
        if node.left is not None:

         # if the checked value is less than the current node's value
            # return the
result of containsHelper for left node
            if value < node.value:

return self.containsHelper(node.left, value)

        # if the current nodes' right node
exists
        if node.right is not None:
            # if the checked value is more than the
current node's value
            # return the result of containsHelper for right node

    if value > node.value:
                return self.containsHelper(node.right,
value)

        # returns False if all "True" conditions are not met

return False
        pass



    def getHeight(self, node):
        # YOUR CODE STARTS
HERE
        # call and return the results from getHeightHelper for the node
        return
self.getHeightHelper(node)
        pass


    def getHeightHelper(self, node):
        #
if the node is empty return zero
        if node is None:
            return 0

        #
declare the two variables for storing the height for both left and right sides

height_left_subtree = 0
        height_right_subtree = 0
        # call getHeightHelper for
left node is it exist
        # store the result in height_left_subtree and plus one

if node.left is not None:
            height_left_subtree = self.getHeightHelper(node.left) +
1
        # call getHeightHelper for right node is it exist
        # store the result in
height_right_subtree and plus one
        if node.right is not None:

height_right_subtree = self.getHeightHelper(node.right) + 1
```

```
        # return the higher
value of the two between height_left_subtree and height_right_subtree
        return
max(height_left_subtree, height_right_subtree)



    def get_closest(self, item):

# YOUR CODE STARTS HERE
        # if the tree is Empty return None
        if
self.isEmpty():
            return None

        # call and return the result from
get_closest_helper
        return self.get_closest_helper(item, self.root.value, self.root)

      pass

    # item is the item to find
    # closest is the current closest value in the
tree
    # node is the current node
    def get_closest_helper(self, item, closest,
node):

        # if current node is Node return the closest value
        if node is
None:
            return closest
        # if the difference between the item and closest is
larger than the difference between item and the value of
        # the current node, change
the value of closes to the current node's value
        if abs(item - closest) > abs(item -
node.value):
            closest = node.value
        # if the item is larger than the
current node's value, store the result of get_closest_helper for right node
        if item
> node.value:
            closest = self.get_closest_helper(item, closest, node.right)

    # if the item is smaller than the current node's value, store the result of
get_closest_helper for left node
        if item < node.value:
            closest =
self.get_closest_helper(item, closest, node.left)

        # return closest if all the above
conditions are not met
        return closest
        pass
```