```
class TreeNode:
    def __init__(self, element, parent=None, left=None, right=None):
        self._element = element
        self._parent = parent
        self._left = left
        self._right = right

    def element(self):
        return self._element

    def setElement(self, element):
        self._element = element

    def parent(self):
        return self._parent

    def setParent(self, p):
        self._parent = p

    def left(self):
        return self._left

    def setLeft(self, p):
        self._left = p

    def right(self):
        return self._right

    def setRight(self, p):
        self._right = p

    def numChildren(self):
        if self.left() is None and self.right() is None:
            return 0
        elif ((self.right() is not None and self.left()) is None
              or (self.left() is not None and self.right() is None)):
            return 1
        else:
            return 2

    def isLeaf(self):
        return self.numChildren() == 0

    def height(self):
        if self is None or self.isLeaf():
            return 0

        elif self.left() is None:
            return 1 + self.right().height()
        elif self.right() is None:
```

```
            return 1 + self.left().height()
        else:
            return 1 + max(self.left().height(), self.right().height())

    def sibling(self):
        if self.parent() is None:
            return None
        elif self.parent().left() == self:
            return self.parent().right()
        else:
            return self.parent().left()

    def visit(self):
        print(self.element(), end=' ')

    def inorder(self):
        if self is not None:
            if self.left() is not None:
                self.left().inorder()
            self.visit()
            if self.right() is not None:
                self.right().inorder()

    def insert(self, p):
        if p.element() < self.element():
            if self.left() is None:
                self.setLeft(p)
                p.setParent(self)
            else:
                self.left().insert(p)
        else:
            if self.right() is None:
                self.setRight(p)
                p.setParent(self)
            else:
                self.right().insert(p)


class LinkedBinarySearchTree:
    def __init__(self):
        self._root = None
        self._size = 0

    def __len__(self):
        return self._size

    def root(self):
        return self._root

    def isEmpty(self):
```

```
    return len(self) == 0

def isRoot(self, p):
    return p == self.root()

def addRoot(self, e):
    if self.root() is not None:
        raise ValueError('Root exists.')

    else:
        self._size = 1
        self._root = TreeNode(e)
        return self.root()

def depth(self, p):
    depthOfNode = 0
    pointer = p
    while pointer.parent() is not None:
        pointer = pointer.parent()
        depthOfNode += 1
    return depthOfNode

def height(self):
    return self.root().height()

### Mutators ###

def insert(self,e):
    if self.isEmpty():
        self.addRoot(e)
    else:
        p = TreeNode(e)
        self.root().insert(p)
        self._size += 1

def inorder(self):
    if not self.isEmpty():
        self.root().inorder()

def find(self, element):

    #
    # Executes a search of the Binary Search Tree beginning at self.root()
    # and continuing until element is found, or the search runs off the end
    # of a leaf.
    #
    # self The implicit parameter for the Tree
    #
    # Returns a list with 2 values – the first value is True if element was
    #                                found, False otherwise -
```

```
        #                                the second value is the number of
locations
        #                                visited during the search.
        #
        visit = 0
        if self.isEmpty():
            return None
        else:
            pointer = self.root()
            while pointer is not None:
                visit += 1
                if pointer.element() > element:
                    pointer = pointer.left()
                elif pointer.element() < element:
                    pointer = pointer.right()
                else:
                    return [True,visit]

            return [False,visit]
```