James Davidson – 7815601

# Question 2 Requires the Graphviz module

To install: $ pip install graphviz

GitHub Link

https://github.coventry.ac.uk/davids44/210CT-Coursework/

## Question 1

```
import re
from pathlib import Path


class Node(object):
    """ Class creates a node with a value and connections left and right """

    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None


def tree_insert(item, tree=None):
    """
    item - String
    tree - object
    If tree is None a new tree is created
    Inserts 'item' in the tree 'tree'

    returns tree
    """

    item = item.lower()

    if tree is None:
        tree = Node(item)

    else:
        if item < tree.value:
            if tree.left is None:
                tree.left = Node(item)
            else:
                tree_insert(item, tree.left)
        else:
            if tree.right is None:
                tree.right = Node(item)
            else:
                tree_insert(item, tree.right)
    return tree


def tree_find(item, tree=None, verbose=True):
    """
    item - String
    tree - object

    Finds 'item' in tree 'tree'

    returns Whether it found the 'item'
    """

    item = item.lower()

    if tree.value is None:
        print("No tree given")
        return False
    found = "No"

    def findrecurse(tree):
        """
        tree - object

        recursive calls to find item in tree

        returns "Yes" or "No"
```

```
        """
        if tree.value == item:
            if verbose:
                print("Item found")
            nonlocal found
            found = "Yes"
            return tree
        elif tree.value > item:
            if verbose:
                print(item + " is less than '" + tree.value + "' Going left")
            if tree.left is None:
                return False
            else:
                return findrecurse(tree.left)
        elif tree.value < item:
            if verbose:
                print(item + " is more than '" + tree.value + "' Going right")
            if tree.right is None:
                return False
            else:
                return findrecurse(tree.right)
        return False

    node = findrecurse(tree)

    if found == "No":
        if verbose:
            print("'" + item + "' was not found in the tree!")
        return False
    return node


def tree_count(tree):
    """
    tree - object
    data - list
    count - dictionary

    returns count
    """
    data = printinorder(tree)
    count = {}

    def countrecurse(word):
        if word in count:
            count[word] += 1
        else:
            count[word] = 1

    for word in data:
        # Loops through all words in data
        countrecurse(word)
    return count


def createtree(itemlist=None):
    """
    itemlist - list
    t - tree - object

    returns tree
    """
    if itemlist is None:
        return False
    for i in range(0, len(itemlist)):
        if i == 0:
            t = tree_insert(itemlist[i])
        else:
```

```
            tree_insert(itemlist[i], t)

    return t


def printpreorder(tree):
    """
    tree - object
    data - list

    returns data - in pre order
    """
    data = []

    def prerecurse(node):
        if not node:
            return
        data.append(node.value)
        prerecurse(node.left)
        prerecurse(node.right)

    prerecurse(tree)
    return data


def printpostorder(tree):
    """
    tree - object
    data - list

    returns data - in post order
    """
    data = []

    def postrecurse(node):
        if not node:
            return
        postrecurse(node.left)
        postrecurse(node.right)
        data.append(node.value)

    postrecurse(tree)
    return data


def printinorder(tree):
    """
    tree - object
    data - list

    returns data - in order
    """
    data = []

    def inrecurse(node):
        if not node:
            return
        inrecurse(node.left)
        data.append(node.value)
        inrecurse(node.right)

    inrecurse(tree)
    return data


def readwords(file):
    """
    Reads in a paragraph from Words.txt
```

```
    Then removes punctuation and splits it in to a list

    returns a list of words

    """
    if not Path(file).is_file():
        # If File is not found
        return None

    with open(file, "r") as f:
        inputwords = f.read()
        # Regex, Removes all non whitespace and non word characters, from the
string inputwords
        inputwords = re.sub(r'[^\w\s]', '', inputwords)
        inputwords = inputwords.lower()

    wordslist = inputwords.split()

    # print(wordslist)
    return wordslist


def gendot(node):
    """
    Generates dot code used by GraphViz to generate visual representation of and
tree given
    Software available here: https://graphviz.gitlab.io/
    Full tree generated 'Full_Tree.png'
    :param node: tree defined by class Node
    :return: dotttext - string
    """
    seq = 0

    def gencode(node, parentseq=None, lorr=None):
        """

        :param node: tree defined by class Node
        :param parentseq: integer - parent node for connections
        :param lorr: string - if the connections to parent was from the left or
right
        :return: string - generated dot code
        """
        nonlocal seq
        seq += 1

        dot = ""
        # Creates the node
        dot += "node" + str(seq) + '[label="' + str(node.value) + '"];'

        if parentseq is not None:
            dot += "node" + str(parentseq) + ' -> ' + "node" + str(seq) +
'[label="' + str(lorr) + '"];'

        thisseq = seq
        # Creates the connection
        if node.left is not None:
            dot += gencode(node.left, thisseq, "L")

        if node.right is not None:
            dot += gencode(node.right, thisseq, "R")

        return dot

    dottext = 'digraph { node [fixedsize=true, fontname="Arial", fontsize=8];'

    dottext += gencode(node)

    dottext += "}"
```

```
    return dottext


if __name__ == "__main__":
    words = readwords("Words.txt")
    bintree = createtree(words)
    print("Pre order ", printpreorder(bintree))
    print("Post order ", printpostorder(bintree))
    print("In order ", printinorder(bintree))
    print(tree_count(bintree))
    print(tree_find("word", bintree).value)
    print("\n" + str(gendot(bintree)))
```

## Question 1 Testing

```
from unittest import TestCase


class TestTree_Question1(TestCase):

    def test_readwords(self):
        from Question1 import readwords
        expected = ['to', 'make', 'your', 'document', 'look', 'professionally',
'produced', 'word', 'provides',
                    'header', 'footer', 'cover', 'page', 'and', 'text', 'box',
'designs', 'that', 'complement',
                    'each', 'other', 'for', 'example', 'you', 'can', 'add', 'a',
'matching', 'cover', 'page', 'header',
                    'and', 'sidebar', 'click', 'insert', 'and', 'then', 'choose',
'the', 'elements', 'you', 'want',
                    'from', 'the', 'different', 'galleries']
        self.assertEqual(readwords("Words.txt"), expected)

    def test_tree_create(self):
        from Question1 import createtree, readwords, printpreorder
        words = readwords("Words.txt")
        bintree = createtree(words)
        expected = ['to', 'make', 'document', 'cover', 'and', 'add', 'a', 'box',
'and', 'and', 'complement', 'can',
                    'click', 'choose', 'designs', 'cover', 'different', 'look',
'header', 'footer', 'each', 'example',
                    'elements', 'for', 'from', 'galleries', 'header', 'insert',
'professionally', 'produced', 'page',
                    'other', 'matching', 'page', 'provides', 'text', 'sidebar',
'that', 'then', 'the', 'the', 'your',
                    'word', 'want', 'you', 'you']
        self.assertEqual(printpreorder(bintree), expected)

    def test_tree_print_post_order(self):
        from Question1 import createtree, readwords, printpostorder
        words = readwords("Words.txt")
        bintree = createtree(words)
        expected = ['a', 'add', 'and', 'and', 'choose', 'click', 'can',
'complement', 'box', 'and', 'cover',
                    'different', 'designs', 'cover', 'elements', 'example', 'each',
'galleries', 'from', 'for',
                    'footer', 'insert', 'header', 'header', 'look', 'document',
'matching', 'other', 'page', 'page',
                    'produced', 'sidebar', 'the', 'the', 'then', 'that', 'text',
'provides', 'professionally', 'make',
                    'want', 'you', 'you', 'word', 'your', 'to']
        self.assertEqual(printpostorder(bintree), expected)

    def test_tree_print_in_order(self):
        from Question1 import createtree, readwords, printinorder
        words = readwords("Words.txt")
        bintree = createtree(words)
        expected = ['a', 'add', 'and', 'and', 'and', 'box', 'can', 'choose',
'click', 'complement', 'cover', 'cover',
                    'designs', 'different', 'document', 'each', 'elements',
'example', 'footer', 'for', 'from',
                    'galleries', 'header', 'header', 'insert', 'look', 'make',
'matching', 'other', 'page', 'page',
                    'produced', 'professionally', 'provides', 'sidebar', 'text',
'that', 'the', 'the', 'then', 'to',
                    'want', 'word', 'you', 'you', 'your']
        self.assertEqual(printinorder(bintree), expected)

    def test_find_in(self):
        from Question1 import createtree, readwords, tree_find
        words = readwords("Words.txt")
```

```
        bintree = createtree(words)
        expected = "look"
        self.assertEqual(tree_find("look", bintree, True).value, expected)

    def test_find_missing(self):
        from Question1 import createtree, readwords, tree_find
        words = readwords("Words.txt")
        bintree = createtree(words)
        expected = False
        try:
            actual = tree_find("missing", bintree, True).value
        except AttributeError:
            actual = False
        self.assertEqual(actual, expected)

    def test_tree_count(self):
        from Question1 import createtree, readwords, tree_count
        words = readwords("Words.txt")
        bintree = createtree(words)
        expected = {'a': 1, 'add': 1, 'and': 3, 'box': 1, 'can': 1, 'choose': 1,
'click': 1, 'complement': 1, 'cover': 2, 'designs': 1, 'different': 1, 'document':
1, 'each': 1, 'elements': 1, 'example': 1, 'footer': 1, 'for': 1, 'from': 1,
'galleries': 1, 'header': 2, 'insert': 1, 'look': 1, 'make': 1, 'matching': 1,
'other': 1, 'page': 2, 'produced': 1, 'professionally': 1, 'provides': 1,
'sidebar': 1, 'text': 1, 'that': 1, 'the': 2, 'then': 1, 'to': 1, 'want': 1,
'word': 1, 'you': 2, 'your': 1}

        self.assertEqual(tree_count(bintree), expected)
```

## Question 2

```
from Question1 import readwords, tree_find, printinorder, gendot


class Node(object):
    """
    Class creates a node with a value and connections left and right.
    Modified from Question to include the nodes parent
    """

    def __init__(self, value, parent=None):
        self.value = value
        self.left = None
        self.right = None
        self.parent = parent


def tree_insert(item, tree=None):
    """
    Modified from Question 1 to include the parent tree
    item - String
    tree - object
    If tree is None a new tree is created
    Inserts 'item' in the tree 'tree'

    returns tree
    """
    if tree is None:
        tree = Node(item)

    else:
        if item < tree.value:
            if tree.left is None:
                tree.left = Node(item, tree)
            else:
                tree_insert(item, tree.left)
        else:
            if tree.right is None:
                tree.right = Node(item, tree)
            else:
                tree_insert(item, tree.right)
    return tree


def createtree(itemlist=None):
    """
    itemlist - list
    t - tree - object

    returns tree
    """
    if itemlist is None:
        return False
    for i in range(0, len(itemlist)):
        if i == 0:
            t = tree_insert(itemlist[i])
        else:
            tree_insert(itemlist[i], t)

    return t


def delete_node(node):
    """
    Deletes the given node from a tree
    :param node: tree defined by class Node
    :return: Bool if it deleted the node
```

```
    """

    # if the node as two children
    if node.left is not None and node.right is not None:
        # Takes the smallest item from the right sub-tree
        # Swaps the value, then deletes that node
        minnode = find_min_node(node.right)

        node.value = minnode.value

        delete_node(minnode)

        return True
    # if the node has no children
    if node.left is None and node.right is None:
        # Sets the parent.left or parent.right to None, depending on what side the
node is on
        try:
            if node.parent.left.value == node.value:
                node.parent.left = None
        except AttributeError:
            pass

        try:
            if node.parent.right.value == node.value:
                node.parent.right = None
        except AttributeError:
            pass

        return True

    # if the node only has one child to the left
    if node.left is not None:
        try:
            if node.parent.left.value == node.value:
                node.parent.left = node.left
        except AttributeError:
            pass

        try:
            if node.parent.right.value == node.value:
                node.parent.right = node.left
        except AttributeError:
            pass

        return True

    # if the node only has one child to the right
    if node.right is not None:
        try:
            if node.parent.left.value == node.value:
                node.parent.left = node.right
        except AttributeError:
            pass

        try:
            if node.parent.right.value == node.value:
                node.parent.right = node.right
        except AttributeError:
            pass

        return True

    return False


def find_min_node(node):
    """
```

```
    Find the smallest item in a tree
    :param node: tree defined by class Node
    :return: node
    """
    # if there is a node to the left, go left else return the node
    if node.left is None:
        return node

    return find_min_node(node.left)


if __name__ == "__main__":
    words = readwords("Words.txt")
    bintree = createtree(words)
    nodetodelete = tree_find("box", bintree, False)
    if nodetodelete is False:
        print("Word not found")
    else:
        print("In order before deletion \n", printinorder(bintree))
        delete_node(nodetodelete)
        print("In order after deleting the word box \n", printinorder(bintree))
```

## Question 2 Testing

```
from unittest import TestCase


class Test_Question_2(TestCase):

    def test_find_min_node(self):
        # Finds the left most node from a subtree
        from Question2 import readwords, createtree, tree_find, find_min_node

        words = readwords("Words.txt")
        bintree = createtree(words)

        item = tree_find("other", bintree, False)

        minnode = find_min_node(item)

        self.assertEqual('matching', minnode.value)

    def test_delete_node_no_children(self):
        # Removing "a"
        print("\n\n Removing 'a', has no children")
        from Question2 import delete_node, createtree, readwords, printinorder,
tree_find, gendot

        expected = ['add', 'and', 'and', 'and', 'box', 'can', 'choose', 'click',
'complement', 'cover', 'cover',
                    'designs', 'different', 'document', 'each', 'elements',
'example', 'footer', 'for', 'from',
                    'galleries', 'header', 'header', 'insert', 'look', 'make',
'matching', 'other', 'page', 'page',
                    'produced', 'professionally', 'provides', 'sidebar', 'text',
'that', 'the', 'the', 'then', 'to',
                    'want', 'word', 'you', 'you', 'your']

        words = readwords("Words.txt")
        bintree = createtree(words)
        nodetodelete = tree_find("a", bintree, False)
        if nodetodelete is False:
            self.assertFalse(True)
        else:
            print("In order before deletion \n", printinorder(bintree))
            print("Original\n" + gendot(bintree))
            delete_node(nodetodelete)
            print("In order after deleting the word 'box' \n",
printinorder(bintree))
            print("After deletion\n" + gendot(bintree))
            self.assertEqual(expected, printinorder(bintree))

    def test_delete_node_one_child_left(self):
        # Removing "your" only has a left child
        print("\n\n Removing 'your', has one child")
        from Question2 import delete_node, createtree, readwords, printinorder,
tree_find, gendot

        expected = ['a', 'add', 'and', 'and', 'and', 'box', 'can', 'choose',
'click', 'complement', 'cover', 'cover',
                    'designs', 'different', 'document', 'each', 'elements',
'example', 'footer', 'for', 'from',
                    'galleries', 'header', 'header', 'insert', 'look', 'make',
'matching', 'other', 'page', 'page',
                    'produced', 'professionally', 'provides', 'sidebar', 'text',
'that', 'the', 'the', 'then', 'to',
                    'want', 'word', 'you', 'you']

        words = readwords("Words.txt")
        bintree = createtree(words)
```

```
        nodetodelete = tree_find("your", bintree, False)
        if nodetodelete is False:
            try:
                self.fail(None)
            except AssertionError:
                pass
        else:
            print("In order before deletion \n", printinorder(bintree))
            print("Original\n" + gendot(bintree))
            print()
            delete_node(nodetodelete)
            print("In order after deleting the word box \n", printinorder(bintree))
            print("After deletion\n" + gendot(bintree))
            self.assertEqual(expected, printinorder(bintree))

    def test_delete_node_one_child_right(self):
        # Removing "provides" only has a right child
        print("\n\n Removing 'provides', has one child")
        from Question2 import delete_node, createtree, readwords, printinorder,
tree_find, gendot

        expected = ['a', 'add', 'and', 'and', 'and', 'box', 'can', 'choose',
'click', 'complement', 'cover', 'cover',
                    'designs', 'different', 'document', 'each', 'elements',
'example', 'footer', 'for', 'from',
                    'galleries', 'header', 'header', 'insert', 'look', 'make',
'matching', 'other', 'page', 'page',
                    'produced', 'professionally', 'sidebar', 'text', 'that', 'the',
'the', 'then', 'to',
                    'want', 'word', 'you', 'you', 'your']

        words = readwords("Words.txt")
        bintree = createtree(words)
        nodetodelete = tree_find("provides", bintree, False)
        if nodetodelete is False:
            try:
                self.fail(None)
            except AssertionError:
                pass
        else:
            print("In order before deletion \n", printinorder(bintree))
            print("Original\n" + gendot(bintree))
            delete_node(nodetodelete)
            print("In order after deleting the word box \n", printinorder(bintree))
            print("After deletion\n" + gendot(bintree))
            self.assertEqual(expected, printinorder(bintree))

    def test_delete_node_two_children(self):
        # Removing "document" has two children
        print("\n\n Removing 'document', has two child")
        from Question2 import delete_node, createtree, readwords, printinorder,
tree_find, gendot

        expected = ['a', 'add', 'and', 'and', 'and', 'box', 'can', 'choose',
'click', 'complement', 'cover', 'cover',
                    'designs', 'different', 'each', 'elements', 'example',
'footer', 'for', 'from',
                    'galleries', 'header', 'header', 'insert', 'look', 'make',
'matching', 'other', 'page', 'page',
                    'produced', 'professionally', 'provides', 'sidebar', 'text',
'that', 'the', 'the', 'then', 'to',
                    'want', 'word', 'you', 'you', 'your']

        words = readwords("Words.txt")
        bintree = createtree(words)
        nodetodelete = tree_find("document", bintree, False)
        if nodetodelete is False:
            try:
```

```
                self.fail(None)
            except AssertionError:
                pass
        else:
            print("In order before deletion \n", printinorder(bintree))
            print("Original\n" + gendot(bintree))
            delete_node(nodetodelete)
            print("In order after deleting the word box \n", printinorder(bintree))
            print("After deletion\n" + gendot(bintree))
            self.assertEqual(expected, printinorder(bintree))
```

## Question 3

```
from graphviz import Graph


def addnode(g, item, connections=None):
    """
    Adds a node to a graph
    :param g: graph - dictionary[item] = [connections]
    :param item: integer - item to be added
    :param connections: list/integer - connections from item to be added
    :return: new graph
    """
    g[item] = []
    # if only one connection is given as an integer
    if connections is int:
        connections = [connections]
    # if there are no connections
    if connections is [] or None:
        g[item] = None
        return g
    # adding connections
    if connections is not None:

        for i in range(0, len(connections)):
            if i == 0:
                g[item] = [connections[i]]
            else:
                g[item].append(connections[i])

    return g


def ispath(g, v1, v2, p=[], d=0):
    """
    Checks for a path from v1 to v2
    :param g: dictionary - Graph
    :param v1: integer - Start node
    :param v2: integer - destination node
    :param p: list - path
    :param d: integer - Stores the current depth of recursion
    :return: list - Path / None
    """
    d += 1
    p.append(v1)
    # If the destination node is found
    if v1 == v2:
        return p
    # else
    if v1 not in g:
        return None
    if v2 not in g:
        return None

    # loops for every node in the connections of the current node
    for connection in g[v1]:
        if connection not in p:
            # Goes to loop through all the connections of this node
            newpath = ispath(g, connection, v2, p, d)

            if newpath:
                if d == 1:
                    # print to file
                    with open("Path.txt", "a") as f:
                        f.write(str(p) + " Path for connecting " + str(v1) + " to "
+ str(v2) + "\n")
                return newpath
    if d == 0:
```

```
        print("No path found")
    return None


def graphvizcode(g):
    """
    Generates dot code for the given graph
    :param g: dictionary - graph
    :return: None
    """
    found = []
    print("graph {", end="")
    for node in g:
        for connection in g[node]:
            if not ([[node], [connection]] in found or [[connection], [node]] in
found):
                print(str(node) + " -- " + str(connection) + ";", end="")
                found.append([[connection], [node]])
    print("}")


def pygraph(g):
    """
    Uses the python graphviz module to generate a given graph
    :param g: dictionary - graph
    :return: None
    """
    found = []
    dot = Graph()

    for nodes in g:
        dot.node(str(nodes))

    for node in g:
        for connection in g[node]:
            if not ([[node], [connection]] in found or [[connection], [node]] in
found):
                dot.edge(str(connection), str(node))
                found.append([[connection], [node]])
    dot.render(view=True)


if __name__ == "__main__":
    """
    g = Dictionary key=item [Connections]
    item = +ve integer
    """

    g = {}

    g = addnode(g, 0, [0, 1, 7])
    g = addnode(g, 1, [0, 2, 6])
    g = addnode(g, 2, [1, 3, 4, 9])
    g = addnode(g, 3, [2])
    g = addnode(g, 4, [2, 5])
    g = addnode(g, 5, [4])
    g = addnode(g, 6, [1, 7, 8])
    g = addnode(g, 7, [0, 6])
    g = addnode(g, 8, [6])
    g = addnode(g, 9, [2])

    for node in g:
        print(node, ": ", g[node])

    print(ispath(g, 3, 8))

    graphvizcode(g)
```

## Question 3 Testing

```
from unittest import TestCase


class Test_is_Path(TestCase):
    def test_add_node(self):
        from Question3 import addnode
        g = {}

        addnode(g, 2)
        self.assertEqual(g, {2: []})

        addnode(g, 3, [2])
        self.assertEqual(g, {2: [], 3: [2]})

        addnode(g, 1, [2, 3])
        self.assertEqual(g, {2: [], 3: [2], 1: [2, 3]})

    def test_is_path(self):
        from Question3 import addnode, ispath

        g = {}

        g = addnode(g, 0, [1])
        g = addnode(g, 1, [2, 6])
        g = addnode(g, 2, [1, 3, 4, 9])
        g = addnode(g, 3, [2])
        g = addnode(g, 4, [2, 5])
        g = addnode(g, 5, [4])
        g = addnode(g, 6, [1, 8])
        g = addnode(g, 7, [6])
        g = addnode(g, 8, [6])
        g = addnode(g, 9, [2])

        self.assertEqual([3, 2, 1, 6, 8], ispath(g, 3, 8))

    def test_is_path_none(self):
        from Question3 import addnode, ispath

        g = {}

        g = addnode(g, 0, [])
        g = addnode(g, 1, [2, 6])
        g = addnode(g, 2, [1, 3, 4, 9])
        g = addnode(g, 3, [2])
        g = addnode(g, 4, [2, 5])
        g = addnode(g, 5, [4])
        g = addnode(g, 6, [1, 8])
        g = addnode(g, 7, [6])
        g = addnode(g, 8, [6])
        g = addnode(g, 9, [2])

        self.assertEqual(None, ispath(g, 4, 0))
```

## Question 4

```
from Question3 import addnode, graphvizcode,  pygraph
import random


def isConnected(g, s=None):
    """
    Checks if all the nodes of a given graph are connected
    :param g: graph - dictionary[item] = [connections]
    :param s: Starting node to check from
    :return: "Yes" or "No"
    """
    # If not starting node is given a random one is chosen
    if s is None:
        s = random.choice(list(g.keys()))
    found = [s]

    def connections(c):
        nonlocal found
        for connection in g[c]:
            if connection in found:
                continue
            else:
                found.append(connection)
                connections(connection)
    connections(s)
    if len(found) == len(g):
        return "Yes"
    else:
        return "No"


def graphvizconnectioncode(g):
    """
    Creates a Directed graph using GraphViz
    If the graph is connected then there will be an arrow to and from each
connection
    :param g: graph - dictionary[item] = [connections]
    :return: None
    """
    found = []
    print("digraph {", end="")
    for node in g:
        for connection in g[node]:
            print(str(node) + " -> " + str(connection) + ";", end="")
            found.append([[connection], [node]])
    print("}")


if __name__ == "__main__":
    g = {}

    g = addnode(g, 0, [0, 1, 7])
    g = addnode(g, 1, [0, 2, 6])
    g = addnode(g, 2, [1, 3, 4, 9])
    g = addnode(g, 3, [2])
    g = addnode(g, 4, [2, 5])
    g = addnode(g, 5, [4])
    g = addnode(g, 6, [1, 7, 8])
    g = addnode(g, 7, [0, 6])
    g = addnode(g, 8, [6])
    g = addnode(g, 9, [2])

    print(isConnected(g))

    graphvizcode(g)
```

```
    graphvizconnectioncode(g)
    # pygraph(g)
```

## Question 4 Testing

```
from unittest import TestCase


class test_is_question4(TestCase):
    def test_is_strongly_connected_true(self):
        from Question4 import isConnected, addnode

        g = {}

        g = addnode(g, 0, [0, 1, 7])
        g = addnode(g, 1, [0, 2, 6])
        g = addnode(g, 2, [1, 3, 4, 9])
        g = addnode(g, 3, [2])
        g = addnode(g, 4, [2, 5])
        g = addnode(g, 5, [4])
        g = addnode(g, 6, [1, 7, 8])
        g = addnode(g, 7, [0, 6])
        g = addnode(g, 8, [6])
        g = addnode(g, 9, [2])

        self.assertEqual(isConnected(g, 0), "Yes")

    def test_is_strongly_connected_false(self):
        from Question4 import isConnected, addnode

        g = {}

        g = addnode(g, 0, [1, 2])
        g = addnode(g, 1, [0, 2])
        g = addnode(g, 2, [0, 1])

        g = addnode(g, 4, [5, 6])
        g = addnode(g, 5, [4, 6])
        g = addnode(g, 6, [4, 5, 6])

        self.assertEqual(isConnected(g, 0), "No")
        self.assertEqual(isConnected(g, 4), "No")
```

## Question 5

```
from Question3 import addnode


def dfs(g, v1, visit=None):
    """
    Depth first traversal of the graph
    :param g: graph - dictionary[item] = [connections]
    :param v1: integer - Where to start
    :param visit: list - where has been visited
    :return: visit
    """
    if visit is None:
        visit = []
    visit.append(v1)

    for next in g[v1]:
        if next in visit:
            continue
        else:
            dfs(g, next, visit)

    return visit


def dfstofile(g, v1):
    """
    Outputs dfs() to file dfs.txt
    :param g: graph - dictionary[item] = [connections]
    :param v1: integer - node to start from
    :return: the path found
    """
    output = dfs(g, v1)
    with open("dfs.txt", "a") as f:
        f.write(str(output) + " DFS starting at " + str(v1) + "\n")

    return output


def bfs(g, v1):
    """
    :param g: Graph - dictionary[key] = [connections]
    :param v1: Node to start from
    :return:
    """
    explored = []
    nodestocheck = [v1]
    # creates new dictionary
    depth = {v1: 0}

    visited = [v1]
    # While there are nodes left to check
    while nodestocheck:
        node = nodestocheck.pop(0)
        explored.append(node)
        connections = g[node]

        for connection in connections:
            if connection not in visited:
                nodestocheck.append(connection)
                visited.append(connection)

                depth[connection] = depth[node] + 1

    print(depth)

    with open("bfs.txt", "a") as f:
```

```
        f.write(str(list(depth.keys())) + " BFS starting at " + str(v1) + "\n")

    return list(depth.keys())


if __name__ == "__main__":
    g = {}

    g = addnode(g, 0, [0, 1, 7])
    g = addnode(g, 1, [0, 2, 6])
    g = addnode(g, 2, [1, 3, 4, 9])
    g = addnode(g, 3, [2])
    g = addnode(g, 4, [2, 5])
    g = addnode(g, 5, [4])
    g = addnode(g, 6, [1, 7, 8])
    g = addnode(g, 7, [0, 6])
    g = addnode(g, 8, [6])
    g = addnode(g, 9, [2])

    print("DFS")
    print(dfstofile(g, 0))

    print("\nBFS")
    print(bfs(g, 8))
```

## Question 5 Testing

```
from unittest import TestCase


class test_question5(TestCase):

    def test_dfs(self):
        from Question3 import addnode
        from Question5 import dfstofile

        g = {}

        g = addnode(g, 0, [0, 1, 7])
        g = addnode(g, 1, [0, 2, 6])
        g = addnode(g, 2, [1, 3, 4, 9])
        g = addnode(g, 3, [2])
        g = addnode(g, 4, [2, 5])
        g = addnode(g, 5, [4])
        g = addnode(g, 6, [1, 7, 8])
        g = addnode(g, 7, [0, 6])
        g = addnode(g, 8, [6])
        g = addnode(g, 9, [2])

        self.assertEqual(dfstofile(g, 0), [0, 1, 2, 3, 4, 5, 9, 6, 7, 8])
        self.assertEqual(dfstofile(g, 4), [4, 2, 1, 0, 7, 6, 8, 3, 9, 5])

    def test_bfs(self):
        from Question3 import addnode
        from Question5 import bfs

        g = {}

        g = addnode(g, 0, [0, 1, 7])
        g = addnode(g, 1, [0, 2, 6])
        g = addnode(g, 2, [1, 3, 4, 9])
        g = addnode(g, 3, [2])
        g = addnode(g, 4, [2, 5])
        g = addnode(g, 5, [4])
        g = addnode(g, 6, [1, 7, 8])
        g = addnode(g, 7, [0, 6])
        g = addnode(g, 8, [6])
        g = addnode(g, 9, [2])

        self.assertEqual(bfs(g, 0), [8, 6, 1, 7, 0, 2, 3, 4, 9, 5])
        self.assertEqual(bfs(g, 5), [5, 4, 2, 1, 3, 9, 0, 6, 7, 8])
```

## Question 6

Not completed

```
def addnode(g, item, connections=None):
    """
    Adds a node to a graph
    :param g: graph - dictionary[item] = [connections]
    :param item: integer - item to be added
    :param connections: list/integer - connections from item to be added
    :return: new graph
    """
    g[item] = []
    # if there are no connections
    if connections is []:
        g[item] = None
        return g
    # adding connections
    if connections is not None:

        for i in range(0, len(connections)):
            if i == 0:
                g[item] = [connections[i]]
            else:
                g[item].append(connections[i])

    return g



if __name__ == "__main__":

    # g: graph - dictionary[item] = [[connection],[weight]]

    g = {}

    g = addnode(g, 0, [[1, 4], [7, 8]])
    g = addnode(g, 1, [[0, 4], [2, 8], [7, 11]])
    g = addnode(g, 2, [[1, 8], [3, 7], [5, 4], [8, 2]])
    g = addnode(g, 3, [[2, 7], [4, 9], [5, 14]])
    g = addnode(g, 4, [[3, 9], [5, 10]])
    g = addnode(g, 5, [[2, 4], [3, 14], [4, 10], [6, 2]])
    g = addnode(g, 6, [[5, 2], [7, 1], [8, 6]])
    g = addnode(g, 7, [[0, 8], [1, 11], [6, 1], [8, 7]])
    g = addnode(g, 8, [[2, 2], [6, 6], [7, 7]])

    print(g)

    print(dijsktra(g, 0))
```

## Question 6 Testing

None Question not completed