Lab Assignment 8
Done by Ms Rodsy Tahmid (Id: 20101021)

###
Trees and Graphs

```
#Ans1)
#Task-1  #python code for height of a tree #recursively
# A class that represents an individual node in a
# Binary Tree
class BinaryTree_Node:
    #Constructor to create a new node
    def __init__(self, parent):
        self.parent = parent
        self.left = None
        self.right = None


def height(Node_root):
    # Checking if the binary tree is empty
    if Node_root == None:
        # If TRUE return 0
        return 0
        # Recursively calling height of each node
    # Returning max(leftHeight, rightHeight) at each iteration
    return 1 + max(height(Node_root.left), height(Node_root.right))

###############################
# Driver Code
Node_root = BinaryTree_Node(3)
Node_root.left = BinaryTree_Node(2)
Node_root.right = BinaryTree_Node(5)
Node_root.left.left = BinaryTree_Node(1)
Node_root.left.right = BinaryTree_Node(4)

print("Height of the tree: ",height(Node_root))


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%#

#Ans2)
#Task-2  #python code for the level of node in a binary tree #recursively

# Helper function for Level(). It
# returns level of the data if data is
# present in tree, otherwise returns 0
def Level_(node_n, value, level):
    if node_n is None:
        return 0

    if node_n.parent == value:
        return level
```

```
    bottom_part = Level_(node_n.left,
                         value, level + 1)
    if (bottom_part != 0):
        return bottom_part

    bottom_part = Level_(node_n.right,
                         value, level + 1)
    return bottom_part

# Returns level of given data value
def to_get_Level(node_n, value):
    return Level_(node_n, value, 1)

########################################
for i in range(1, 6):
    level = to_get_Level(Node_root, i)
    if (level):
        print("Level of the given node",i,
              "is", to_get_Level(Node_root, i))
    else:
        print(i, "is not present in tree")


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%#

#Ans3)
#Task-3  #python code for Pre-order tree traversal

# A function to do preorder tree traversal
def PreorderTraversal(Node_r):
    if Node_r != None:

        print(Node_r.parent) #visit self

        PreorderTraversal(Node_r.left) #visit left child

        PreorderTraversal(Node_r.right) #visit right child

########################################
print("Preorder traversal of binary tree is")
PreorderTraversal(Node_root)


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%#

#Ans4)
#Task-4  #python code for In-order tree traversal

# A function to do inorder tree traversal
def InorderTraversal(Node_r):
    if Node_r != None:

        InorderTraversal(Node_r.left) #visit left child
```

```
        print(Node_r.parent), #print

        InorderTraversal(Node_r.right) #visit right child

########################################
print("Inorder traversal of binary tree is")
InorderTraversal(Node_root)


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%#

#Ans5)
#Task-5  #python code for In-order tree traversal

# A function to do postorder tree traversal
def PostorderTraversal(Node_r):
    if Node_r != None:

        PostorderTraversal(Node_r.left) #visit left child

        PostorderTraversal(Node_r.right) #visit right child

        print(Node_r.parent)# print

########################################
print("Postorder traversal of binary tree is")
PostorderTraversal(Node_root)


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%#

#Ans6)
#Task-6 # python code for comparing two trees

# Function to perform inorder traversal
def InorderTraversal(Node_r):
    if Node_r != None:

        InorderTraversal(Node_r.left) #visit left child

        print(Node_r.parent), #print

        InorderTraversal(Node_r.right) #visit right child

# Function to check if two BSTs
# are identical
def Are_Identical(Node_root, Node_root2):
    # Checking if both the trees are empty
    if (Node_root == None and Node_root2 == None):
        return 1

    # If any one of the tree is non-empty
```

```
    # and other is empty, return false
    elif (Node_root != None and Node_root2 == None):
        return 0
    elif (Node_root == None and Node_root2 != None):
        return 0
    else:  # Checking if current data of both trees
        # equal and recursively check for left
        # and right subtrees
        if (Node_root.parent == Node_root2.parent and
                Are_Identical(Node_root.left, Node_root2.left)
                and Are_Identical(Node_root.right, Node_root2.right)):
            return 1
        else:
            return 0

######################################
# Driver Code

#Node_root2 = BinaryTree_Node(3)
#Node_root2.left = BinaryTree_Node(3)
#Node_root2.right = BinaryTree_Node(8)
#Node_root2.left.left = BinaryTree_Node(2)
#Node_root2.left.right = BinaryTree_Node(4)

Node_root2 = BinaryTree_Node(3)
Node_root2.right = BinaryTree_Node(5)
Node_root2.left = BinaryTree_Node(2)
Node_root2.left.right = BinaryTree_Node(4)
Node_root2.left.left = BinaryTree_Node(1)


if (Are_Identical(Node_root, Node_root2)):
    print("Both of the trees are exactly the same")
else:
    print("The two trees are not the same")


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%#

#Ans7)
#Task-7 #python code for creating a new binary tree from a given binary tree

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

# Helper function that allocates
# a new node with the given data
# and None left and right pointers
def createNode(parent2):
    newNode = BinaryTree_Node(0)
    newNode.parent2 = parent2
```

```
    newNode.left = None
    newNode.right = None
    return newNode

#function to print Inorder traversal
def InorderTraversal(Node_r):
    if Node_r == None:
        return
    InorderTraversal(Node_r.left) #visit left child
    print(Node_r.parent2, end=" ") #print
    InorderTraversal(Node_r.right) #visit right child

# copying function takes two trees,
# original tree and a copy tree
# It recurses on both the trees,
# but when original tree recurses on left,
# copy tree recurses on left and
# vice-versa
def copying(Node_r, copy):
    if (Node_r == None):
        copy = None
        return copy

    # Create new copy node
    # from original tree node
    copy = createNode(Node_r.parent2)
    copy.left = copying(Node_r.left,
                    ((copy).right))
    copy.right= copying(Node_r.right,
                    ((copy).left))
    return copy


######################################
# Driver Code
Node_root = createNode(3)
Node_root.left = createNode(2)
Node_root.right = createNode(5)
Node_root.left.left = createNode(1)
Node_root.left.right = createNode(4)

# Print inorder traversal of the input tree
print("Inorder of original tree: ")
InorderTraversal(Node_root)
copy1 = None
copy1 = copying(Node_root, copy1)

# Print inorder traversal of the copy tree
print("\nInorder of copied tree: ")
InorderTraversal(copy1)


#%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%#
```

#Ans8) # in copy

Cse 220

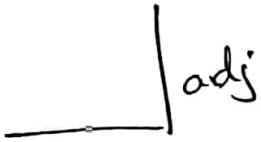
Lab - 8

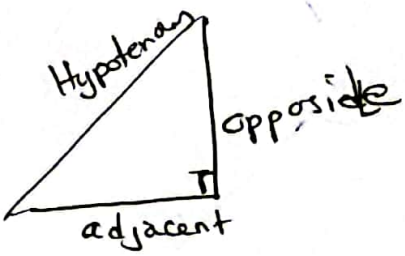
Name : Md Rodry Tahmid
Sec : 12 (Theory)
6 (Lab)
Id : 20101021

Ans 8)

(a)

The ~~at~~ elements of adjacency matrix is indicated that whether pair of vertices are adjacent or not.





Given

Adjacency matrix :

	A	B	C	D	E	F	G	
A	0	1	0	1	1	0	0	A
B	0	0	0	0	0	0	1	B
C	0	1	0	0	0	0	0	C
D	0	0	1	0	0	0	1	D
E	0	0	0	0	0	0	0	E
F	0	0	0	1	0	0	0	F
G	0	0	0	0	1	1	0	G

(8) → continuation. Equivalent Graph: