**Lab Assignment 08**

Name: Moinul Hossain Bhuiyan

Id: 20301002

Section06

**Visualization Of the tree**

print("                  1         ")

print("                ↙   ↘       ")

print("              ↙       ↘     ")

print("           ↙             ↘   ")

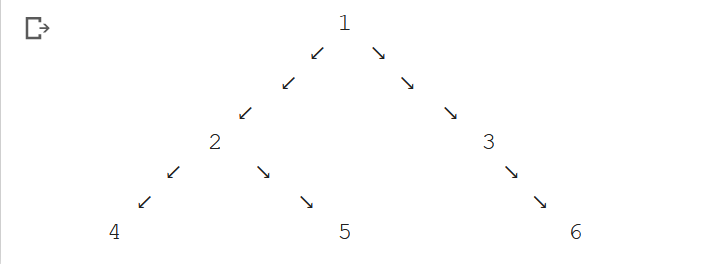
print("         2                  3  ")

print("      ↙     ↘                ↘  ")

print("    ↙          ↘               ↘ ")

print("  4               5               6 ")

**The Output Would be:**

****

**Task01**

class Node(object):

    def \_\_init\_\_(self, c, lft, rht, pnt):

        self.e = None

        self.left = None

        self.right = None

        self.parent = None

        self.e=c

        self.left=lft

        self.right=rht

        self.parent=pnt

def tree(a, i):

    if i<0 or i>=len(a) or a[i] is None:

        return None

    else:

        root = Node(a[i],None,None,None)

        root.left = tree(a,2\*i)

        root.right = tree(a,2\*i+1)

        if root.left is not None:

            root.left.parent = root

        if root.right is not None:

            root.right.parent = root

        return root

    def max(r\_l, r\_r):

        if r\_l> r\_r:

            return r\_l

        return r\_r

def height(root):

    if root is None:

        return 0

    return 1+max(height(root.left),height(root.right))

arr = [None, 1, 2, 3, 4, 5, None, 6]

tree = tree(arr,1)

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

**Output Would Be:**

****

**Task02**

class Node(object):

    def \_\_init\_\_(self, c, lft, rht, pnt):

        self.e = None

        self.left = None

        self.right = None

        self.parent = None

        self.e=c

        self.left=lft

        self.right=rht

        self.parent=pnt

def tree(a, i):

    if i<0 or i>=len(a) or a[i] is None:

        return None

    else:

        root = Node(a[i],None,None,None)

        root.left = tree(a,2\*i)

        root.right = tree(a,2\*i+1)

        if root.left is not None:

            root.left.parent = root

        if root.right is not None:

            root.right.parent = root

        return root

    def max(r\_l, r\_r):

        if r\_l> r\_r:

            return r\_l

        return r\_r

def level(n):

    if n.parent is None:

        return 0

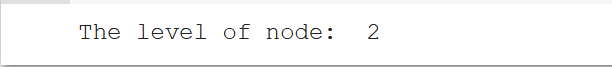
    return 1+level(n.parent)

arr = [None, 1, 2, 3, 4, 5, None, 6]

x = tree(arr,1)

print("The level of node: ",level(x.left.right))

**Output Would Be:**

****

**Task03**

class Node(object):

    def \_\_init\_\_(self, c, lft, rht, pnt):

        self.e = None

        self.left = None

        self.right = None

        self.parent = None

        self.e=c

        self.left=lft

        self.right=rht

        self.parent=pnt

def tree(a, i):

    if i<0 or i>=len(a) or a[i] is None:

        return None

    else:

        root = Node(a[i],None,None,None)

        root.left = tree(a,2\*i)

        root.right = tree(a,2\*i+1)

        if root.left is not None:

            root.left.parent = root

        if root.right is not None:

            root.right.parent = root

        return root

    def max(r\_l, r\_r):

        if r\_l> r\_r:

            return r\_l

        return r\_r

def preordertraversal(r):

    if r is not None:

        print(r.e)

        preordertraversal(r.left)

        preordertraversal(r.right)

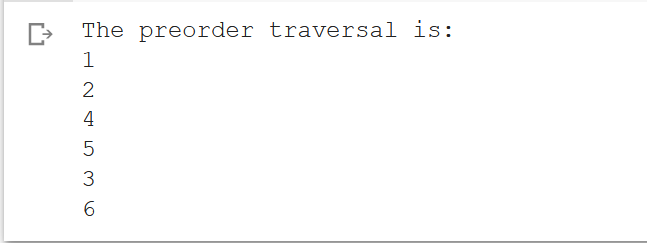
arr = [None, 1, 2, 3, 4, 5, None, 6]

pre = tree(arr,1)

print("The preorder traversal is: ")

preordertraversal(pre)

**Output Would Be:**

****

**Task04**

class Node(object):

    def \_\_init\_\_(self, c, lft, rht, pnt):

        self.e = None

        self.left = None

        self.right = None

        self.parent = None

        self.e=c

        self.left=lft

        self.right=rht

        self.parent=pnt

def tree(a, i):

    if i<0 or i>=len(a) or a[i] is None:

        return None

    else:

        root = Node(a[i],None,None,None)

        root.left = tree(a,2\*i)

        root.right = tree(a,2\*i+1)

        if root.left is not None:

            root.left.parent = root

        if root.right is not None:

            root.right.parent = root

        return root

    def max(r\_l, r\_r):

        if r\_l> r\_r:

            return r\_l

        return r\_r

def inordertraversal(r):

    if r is not None:

        inordertraversal(r.left)

        print(r.e)

        inordertraversal(r.right)

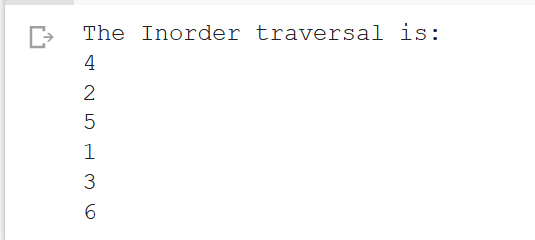
arr = [None, 1, 2, 3, 4, 5, None, 6]

in\_ord = tree(arr,1)

print("The Inorder traversal is: ")

inordertraversal(in\_ord)

**Output Would Be:**

****

**Task05**

class Node(object):

    def \_\_init\_\_(self, c, lft, rht, pnt):

        self.e = None

        self.left = None

        self.right = None

        self.parent = None

        self.e=c

        self.left=lft

        self.right=rht

        self.parent=pnt

def tree(a, i):

    if i<0 or i>=len(a) or a[i] is None:

        return None

    else:

        root = Node(a[i],None,None,None)

        root.left = tree(a,2\*i)

        root.right = tree(a,2\*i+1)

        if root.left is not None:

            root.left.parent = root

        if root.right is not None:

            root.right.parent = root

        return root

    def max(r\_l, r\_r):

        if r\_l> r\_r:

            return r\_l

        return r\_r

def postordertraversal(r):

    if r is not None:

        postordertraversal(r.left)

        postordertraversal(r.right)

        print(r.e)

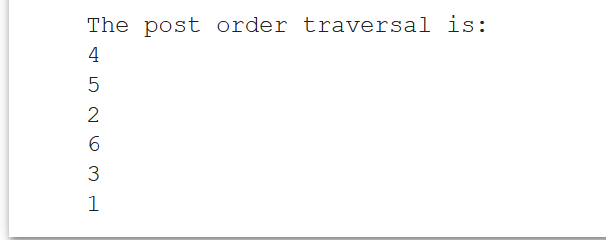
arr = [None, 1, 2, 3, 4, 5, None, 6]

post = tree(arr,1)

print("The post order traversal is: ")

postordertraversal(post)

**Output Would Be:**

****

**Task06**

class Node(object):

    def \_\_init\_\_(self, c, lft, rht, pnt):

        self.e = None

        self.left = None

        self.right = None

        self.parent = None

        self.e=c

        self.left=lft

        self.right=rht

        self.parent=pnt

def tree(a, i):

    if i<0 or i>=len(a) or a[i] is None:

        return None

    else:

        root = Node(a[i],None,None,None)

        root.left = tree(a,2\*i)

        root.right = tree(a,2\*i+1)

        if root.left is not None:

            root.left.parent = root

        if root.right is not None:

            root.right.parent = root

        return root

    def max(r\_l, r\_r):

        if r\_l> r\_r:

            return r\_l

        return r\_r

def smornt(a, b):

    result =""

    i = 0

    while i<len(a):

        j = 0

        while j<len(b):

            if i==j:

                if a[i] is b[j]:

                    result="Same"

                else:

                    result="Not same"

            j += 1

        i += 1

    print(result)

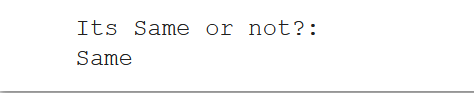
arr\_2 = [None, 1, 2, 3, 4, 5, None, 6]

arr\_3 = [None, 1, 2, 3, 4, 5, None, 6]

print("Its Same or not?:")

smornt(arr\_3,arr\_2)

**Output Would be:**



**Task07**

class Node(object):

    def \_\_init\_\_(self, c, lft, rht, pnt):

        self.e = None

        self.left = None

        self.right = None

        self.parent = None

        self.e=c

        self.left=lft

        self.right=rht

        self.parent=pnt

def tree(a, i):

    if i<0 or i>=len(a) or a[i] is None:

        return None

    else:

        root = Node(a[i],None,None,None)

        root.left = tree(a,2\*i)

        root.right = tree(a,2\*i+1)

        if root.left is not None:

            root.left.parent = root

        if root.right is not None:

            root.right.parent = root

        return root

    def max(r\_l, r\_r):

        if r\_l> r\_r:

            return r\_l

        return r\_r

def copy(a):

    b = [None for \_ in range(len(a))]

    i = 0

    while i<len(a):

        b[i]=a[i]

        i += 1

    return b

arr = [None, 1, 2, 3, 4, 5, None, 6]

print(copy(arr))

**Output Would Be:**

****

**Task08**

Drawing the equivalent graph of the given adjacent matrix:

