**Name: Mohammad Zain Abbas**

**Reg #: 6865**

**DE-36 (CE), Syndicate: A**

**LAB 5 JOURNEL**

**Equipment Used:** Notebook Computer, Python IDLE 3.6

**Lab Tasks:**

1. Implement a LIFO data structure in python.

**SOLUTION CODE:**

class Stack:

def \_\_init\_\_(self):

self.stack = []

def isEmpty(self):

return self.stack == []

def push(self,item):

self.stack.append(item)

def pop(self):

return self.stack.pop()

def top\_Value(self):

return self.stack[len(self.stack)-1]

def size(self):

return len(self.stack)

s = Stack()

print(s.isEmpty())

s.push(4)

s.push('dog')

print(s.top\_Value())

s.push(True)

print(s.size())

print(s.isEmpty())

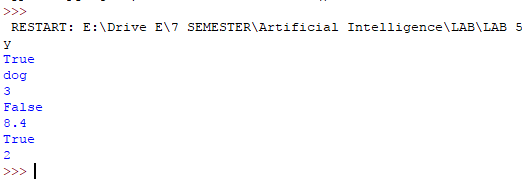
s.push(8.4)

print(s.pop())

print(s.pop())

print(s.size())

**OUTPUT:**



2. Implement DFS algorithm recursively and iteratively in python.

**SOLUTION CODE:**

class Stack:

def \_\_init\_\_(self):

self.stack = []

def isEmpty(self):

return self.stack == []

def push(self,item):

self.stack.append(item)

def pop(self):

return self.stack.pop()

def top\_Value(self):

return self.stack[len(self.stack)-1]

def size(self):

return len(self.stack)

def dfs\_recursive(graph, start\_node, path=[]):

path += [start\_node]

for neighbor in graph[start\_node]:

if neighbor not in path:

path = dfs\_recursive(graph, neighbor, path)

return path

def dfs\_iterative(graph,start\_node):

visited=[];

stack=[start\_node];

while stack:

node=stack.pop();

if node not in visited:

visited.append(node);

print(visited);

for child in reversed(graph[node]):

stack.append(child);

return visited;

def main():

Graph1={'6':['4'],

'4':['3','5','6'],

'3':['2','4'],

'5':['1','2','4'],

'2':['1','3','5'],

'1':['2','5']

}

Graph2={'E':['A','B'],

'A':['B','D','E'],

'B':['A','D','E'],

'D':['A','B','C'],

'C':['D']

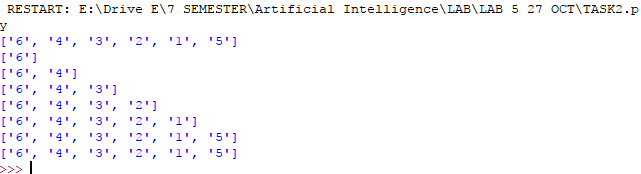
}

print(dfs\_recursive(Graph1,'6'));

print(dfs\_iterative(Graph1,'6'));

main();

**OUTPUT:**



3.Traverse Graph 1 and 2 through implemented DFS algorithm. The starting node is ‘6’ for

Graph 1 while the starting node is ‘E’ for Graph 2.

**SOLUTION CODE:**

class Stack:

def \_\_init\_\_(self):

self.stack = []

def isEmpty(self):

return self.stack == []

def push(self,item):

self.stack.append(item)

def pop(self):

return self.stack.pop()

def top\_Value(self):

return self.stack[len(self.stack)-1]

def size(self):

return len(self.stack)

def dfs\_recursive(graph, start\_node, path=[]):

path += [start\_node]

for neighbor in graph[start\_node]:

if neighbor not in path:

path = dfs\_recursive(graph, neighbor, path)

return path

def main():

Graph1={'6':['4'],

'4':['5','3','6'],

'5':['4','2','1'],

'3':['4','2'],

'2':['3','5','1'],

'1':['5','2']

}

Graph2={'E':['A','B'],

'A':['E','B','D'],

'B':['A','E','D'],

'D':['A','B','C'],

'C':['D']

}

print('For Graph 1');

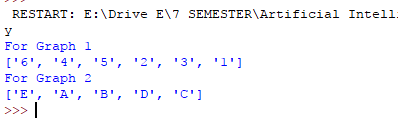
print(dfs\_recursive(Graph1,'6'));

print('For Graph 2');

print(dfs\_recursive(Graph2,'E',path=[]));

main();

**OUTPUT:**



4.Traverse Tree 1 and 2 using Pre-Order, In-Order and Post-Order DFS traversals. The starting node is ‘1’ for Tree 1 while the starting node is ‘Frankfurt’ for Tree 2.

**SOLUTION CODE:**

class Tree():

def \_\_init\_\_(self):

self.left = None

self.right = None

self.data = None

def Inorder(node):

if(node!=None):

Inorder(node.left)

print(node.data)

Inorder(node.right)

def Preorder(node):

if(node!=None):

print(node.data)

Preorder(node.left)

Preorder(node.right)

def Postorder(node):

if(node!=None):

Postorder(node.left)

Postorder(node.right)

print(node.data)

Tree1 = Tree()

Tree1.data = 1

Tree1.left = Tree()

Tree1.left.data = 2

Tree1.right=Tree()

Tree1.right.data = 3

Tree1.left.left = Tree()

Tree1.left.left.data = 4

Tree1.left.right=Tree()

Tree1.left.right.data = 5

Tree1.left.left.left = Tree()

Tree1.left.left.left.data = 6

Tree1.left.right.left=Tree()

Tree1.left.right.left.data = 7

Tree1.left.right.right=Tree()

Tree1.left.right.right.data = 8

print('Inorder Traversion');

Inorder(Tree1);

print('Preorder Traversion');

Preorder(Tree1);

print('Postorder Traversion');

Postorder(Tree1);

Tree2 = Tree()

Tree2.data = 50

Tree2.left = Tree()

Tree2.left.data = 17

Tree2.right=Tree()

Tree2.right.data = 76

Tree2.right.left=Tree()

Tree2.right.left.data = 54

Tree2.right.left.right=Tree()

Tree2.right.left.right.data = 72

Tree2.right.left.right.left=Tree()

Tree2.right.left.right.left.data = 67

Tree2.left.left = Tree()

Tree2.left.left.data = 9

Tree2.left.right=Tree()

Tree2.left.right.data = 23

Tree2.left.right.left=Tree()

Tree2.left.right.left.data = 19

Tree2.left.left.right = Tree()

Tree2.left.left.right.data = 14

Tree2.left.left.right.left=Tree()

Tree2.left.left.right.left.data = 12

print('Inorder Traversion');

Inorder(Tree2);

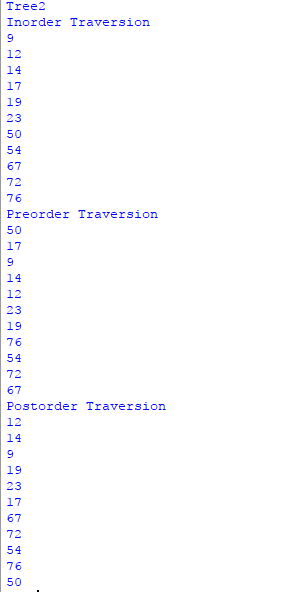
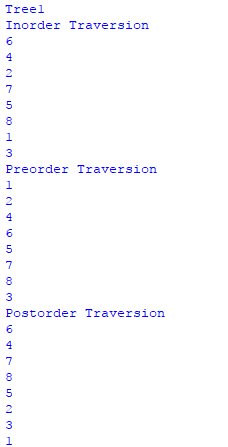
print('Preorder Traversion');

Preorder(Tree2);

print('Postorder Traversion');

Postorder(Tree2);

**OUTPUT:**

**5.** Write a script to decompose the given image into an undirected graph where the pixel represents the vertices and adjacent vertices are connected to each other via 4-connectivity. Use DFS algorithm to traversal decomposed image starting from pixel 150.

|  |  |  |
| --- | --- | --- |
| 150 | 2 | 5 |
| 80 | 145 | 45 |
| 74 | 102 | 165 |

**SOLUTION CODE:**

def dfs\_recursive(graph, start\_node, path=[]):

path += [start\_node]

for neighbor in graph[start\_node]:

if neighbor not in path:

path = dfs\_recursive(graph, neighbor, path)

return path

def image\_graph(image,start):

graph=dict();

for i in range(len(image)):

for j in range(len(image[0])):

graph[image[i][j]]=list();

if(i<len(image)-1):

graph[image[i][j]].append(image[i+1][j]);

if(i>0):

graph[image[i][j]].append(image[i-1][j]);

if(j<len(image)-1):

graph[image[i][j]].append(image[i][j+1]);

if(j>0):

graph[image[i][j]].append(image[i][j-1]);

return dfs\_recursive(graph,start);

def main():

image=[[150,2,5],[80,145,45],[74,102,165]];

print(image\_graph(image,150));

main();

**OUTPUT:**

