**Name: Mohammad Zain Abbas**

**Reg #: 6865**

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

**LAB 4 JOURNEL**

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

**Lab Tasks:**

1. Implement a FIFO data structure in python.

**SOLUTION CODE:**

class Queue:

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

self.queue = []

def isEmpty(self):

return self.queue == []

def enqueue(self, value):

self.queue.insert(0,value)

def dequeue(self):

return self.queue.pop()

def size(self):

return len(self.queue);

def main():

q=Queue();

q.enqueue(4);

q.enqueue(6);

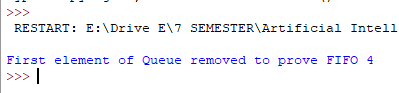
q.enqueue(8);

q.enqueue(1);

print(q.dequeue());

main();

**OUTPUT:**



2. Use the implemented FIFO data structure to implement BFS for Graph 1 and 2 in python. The starting node is ‘6’ for Graph 1 while the starting node is ‘C’ for Graph 2.

**SOLUTION CODE:**

class Queue:

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

self.queue = []

def isEmpty(self):

return self.queue == []

def enqueue(self, value):

self.queue.insert(0,value)

def dequeue(self):

return self.queue.pop()

def size(self):

return len(self.queue)

def Breadth\_First\_Search(graph,start\_node,q=Queue(),visited=[]):

visited.append(start\_node);

print(visited);

neighbours=graph[start\_node];

for i in neighbours:

q.enqueue(i);

while(not(q.isEmpty())):

front=q.dequeue();

if(visited.count(front)==0):

Breadth\_First\_Search(graph,front,q,visited);

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']

}

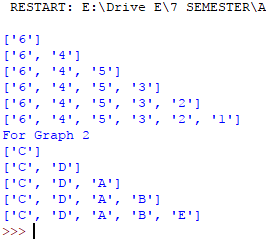
Breadth\_First\_Search(Graph1,'6');

print('For Graph 2');

Breadth\_First\_Search(Graph2,'C',visited=[]);

main();

**OUTPUT:**



3. Implement BFS for Tree 1 and 2 in python. The starting node is ‘1’ for Tree 1 while the starting node is ‘Frankfurt’ for Tree 2.

**SOLUTION CODE:**

import time

class Queue:

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

self.queue=[];

def isEmpty(self):

return len(self.queue)==0;

def Enqueue(self,value):

return self.queue.insert(0,value);

def Dequeue(self):

return self.queue.pop();

def size(self):

return len(self.queue)

class Tree():

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

self.left = None

self.right = None

self.middle=None

self.data = None

#Tree\_numbers

Tree\_numbers = Tree()

Tree\_numbers.data = 1

Tree\_numbers.left = Tree()

Tree\_numbers.left.data = 2

Tree\_numbers.middle=Tree()

Tree\_numbers.middle.data = 3

Tree\_numbers.right = Tree()

Tree\_numbers.right.data = 4

Tree\_numbers.left.left = Tree()

Tree\_numbers.left.left.data = 5

Tree\_numbers.left.middle = Tree()

Tree\_numbers.left.middle.data=6

Tree\_numbers.right.middle=Tree()

Tree\_numbers.right.middle.data=7

Tree\_numbers.right.right=Tree()

Tree\_numbers.right.right.data=8

Tree\_numbers.left.left.left=Tree()

Tree\_numbers.left.left.left.data=9

Tree\_numbers.left.left.middle=Tree()

Tree\_numbers.left.left.middle.data=10

Tree\_numbers.right.middle.middle=Tree()

Tree\_numbers.right.middle.middle.data=11

Tree\_numbers.right.middle.right=Tree()

Tree\_numbers.right.middle.right.data=12

#Tree\_cities

Tree\_cities = Tree()

Tree\_cities.data = 'Frankfurt'

Tree\_cities.left = Tree()

Tree\_cities.left.data = 'Mannheim'

Tree\_cities.middle=Tree()

Tree\_cities.middle.data = 'Wurzburg'

Tree\_cities.right = Tree()

Tree\_cities.right.data = 'kassel'

Tree\_cities.left.middle = Tree()

Tree\_cities.left.middle.data= 'Karlsruhe'

Tree\_cities.middle.left=Tree()

Tree\_cities.middle.left.data='Nurnberg'

Tree\_cities.middle.right=Tree()

Tree\_cities.middle.right.data='Erfurt'

Tree\_cities.right.middle = Tree()

Tree\_cities.right.middle.data = 'Munchen'

Tree\_cities.left.middle.middle = Tree()

Tree\_cities.left.middle.middle.data= 'Augsburg'

Tree\_cities.middle.left.middle=Tree()

Tree\_cities.middle.left.middle.data='Stuttgart'

from collections import deque

def BreadthFirstSearch(root, queue = deque()):

if root is None:

return

print (root.data)

for node in [root.left, root.middle, root.right]:

if node:

queue.append(node);

if queue:

BreadthFirstSearch(queue.popleft(), queue)

print('Tree\_numbers')

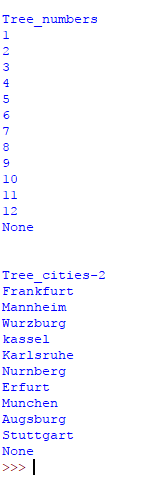
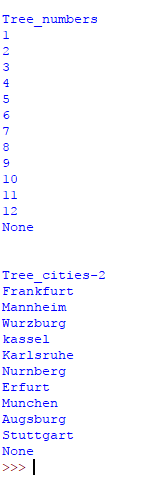
print(BreadthFirstSearch(Tree\_numbers));

print('\n')

print('Tree\_cities-2')

print(BreadthFirstSearch(Tree\_cities));

**OUTPUT:**

**Q:** Decompose the following 3x3 3-bit grayscale image into an undirected graph using 8 connectivity and search all pixels using BFS algorithm starting with node ‘5’. Also tell the total time taken by your algorithm.

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

**SOLUTION CODE:**

class Queue:

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

self.queue = []

def isEmpty(self):

return self.queue == []

def enqueue(self, value):

self.queue.insert(0,value)

def dequeue(self):

return self.queue.pop()

def size(self):

return len(self.queue)

import time

start=time.clock();

def Breadth\_First\_Search(graph,start\_node,q=Queue(),visited=[]):

visited.append(start\_node);

print(visited);

neighbours=graph[start\_node];

for i in neighbours:

q.enqueue(i);

while(not(q.isEmpty())):

front=q.dequeue();

if(visited.count(front)==0):

Breadth\_First\_Search(graph,front,q,visited);

print('Time Taken: ',time.clock()-start)

def main():

Graph\_Image={'2':['5','45','80','145','150'],

'5':['2','45','145'],

'45':['2','5','102','145','165'],

'74':['80','102','145'],

'80':['2','74','102','145','150'],

'102':['45','74','80','145','165'],

'145':['2','5','45','74','80','102','150','165'],

'150':['2','80','145'],

'165':['45','102','145']}

Breadth\_First\_Search(Graph\_Image,'5');

main();

**OUTPUT:**

