Graph Data Structure Introduction

```
# Graphs are data structures similar to trees. They differ
# in that graphs have a shape representative of what they model.

# With graphs nodes are called vertices and the lines connecting
# them are edges. Graphs are considered connected if there is one
# path from each vertex to each other vertex.

# Graphs don't have fixed organization like trees. Trees have
# a maximum of 2 children, but with graphs each vertex can
# connect to any number of other vertices.


# 4. This is the custom stack used with the Depth First Search
class MyStack:
    def __init__(self, size):
        self.size = size
        self.my_stack = [0] * self.size
        self.top = -1

    # Increment the index and insert value
    def push(self, val):
        self.top += 1
        self.my_stack[self.top] = val

    # Decrement the value of top to move down the stack
    # and return the value previously in the top index
    def pop(self):
        self.top -= 1
        return self.my_stack[self.top + 1]

    # Return the top value, but don't delete it
    def peek(self):
        return self.my_stack[self.top]

    # Checks if the stack is empty
    def is_empty(self):
        return self.top == -1


# Each vertex will have a name
class Vertex:
    def __init__(self, name):
        self.name = name
        # Used for searching
        self.visited = False


# Here I'll model a graph using a vertex array
class Graph:
    def __init__(self):
        self.max_vertices = 10
```

```
    self.vertex_list = [0]*10
    # Multidimensional list of zeroes
    # Use a generator to create a list of
    # elements defined by max and assigned 0
    self.adjacency_matrix = [[0] * self.max_vertices for i in range(self.max_vertices)]
    self.vertex_count = 0

    # Used for Depth First Searching (set size of stack to 20)
    self.the_stack = MyStack(20)

def add_vertex(self, name):
    self.vertex_list[self.vertex_count] = Vertex(name)
    self.vertex_count += 1

# We will use an adjacency matrix that defines whether
# an edge lies between 2 vertices
# Each row &column represents a single vertex and
# a 1 lies in a cell if vertices connect
# Example when A connects to B & C but B & C don't
#    A  B  C
# A  0  1  1
# B  1  0  0
# C  1  0  0
def add_edge(self, first, last):
    self.adjacency_matrix[first][last] = 1
    self.adjacency_matrix[last][first] = 1

def print_vertices(self):
    for i in self.vertex_list:
        # If an instance of int print 0
        if isinstance(i, int):
            print(0)
        else:
            print(i.name)

def print_edges(self):
    for row in self.adjacency_matrix:
        for elem in row:
            print(elem, end=' ')
        print()

# 2. We will use Depth First Searching to locate all vertices.
# This works by monitoring vertices that were visited as well as
# adjacent vertices. We visit the 1st vertex in the vertex_list,
# check visited, then work down the column checking those vertices
# that are connected (marked with a 1). When we get to the end of
# the column we move to the next and continue searching.

# 3. This function returns the next unvisited vertex or -1
def get_next_unvisited_vertex(self, curr_vertex):
    for i in range(0, self.vertex_count):
        if self.adjacency_matrix[curr_vertex][i] == 1 and self.vertex_list[i].visited is False:
            return i
    return -1
```

```
    # 5. This is the Depth First Search Function
    def df_search(self):
        # Start searching at vertex in index 0
        self.vertex_list[0].visited = True

        # Print it
        print(self.vertex_list[0].name)

        # Push 0 in the stack
        self.the_stack.push(0)

        while not self.the_stack.is_empty():
            vertex = self.get_next_unvisited_vertex(self.the_stack.peek())

            if vertex == -1:
                self.the_stack.pop()
            else:
                self.vertex_list[vertex].visited = True
                # Print it
                print(self.vertex_list[vertex].name)
                self.the_stack.push(vertex)

        # Stack is empty so set all vertices back to unvisited
        for i in range(0, self.max_vertices):
            if isinstance(i, int):
                pass
            else:
                self.vertex_list[i].visited = False



graph = Graph()
graph.add_vertex("A")
graph.add_vertex("B")
graph.add_vertex("C")
graph.add_vertex("D")
graph.add_vertex("E")
# A connects to B
graph.add_edge(0, 1)
# B connects to C
graph.add_edge(1, 2)
# A connects to D
graph.add_edge(0, 3)
# D connects to E
graph.add_edge(3, 4)
graph.print_vertices()
graph.print_edges()

# 6. Run Depth First Search
graph.df_search()
```