Graph ADT

Typical Graph ADT will include the following operations:
1. graph: create new empty graph
2. addVertex(v): add instance of vertex to the graph
3. addEdge(fV, tV): adds a new directed edge from fV vertex to tV vertex
4. addEdge(fV, tV, weight): adds a new weighted directed edge from fV vertex to tV vertex
5. getVertex(key): finds and returns vertex in the graph for key value
6. removeVertex(v): removes vertex from the graph and all instances of its edges
7. removeEdge(fV, tV): removes edge from the graph
6. getVertices(): returns a list of all vertices in the graph
7. getEdges(): returns a list of all vertices in a graph

The implementations for this operations may differ across programs. Unlike some of the other ADTs there is no standard implementation. Also, how you implement the graph will differ based on what you will use it for.

Adjacency Matrix

If only need to create the graph and there is no separate data for each vertex and do not need to remove any of the vertices, then can implement as **adjacency matrix** and use the array's index as label. So implementation could be something like the below pseudo-code:

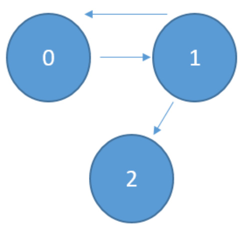
```
class Graph

    createGraph(max)
    adj[][]    # bool: True if exists an edge
    size       # number of vertices  starts at 0


    addVertex()
    size = size + 1

    addEdge(fromV, toV)
    adj[fromV][toV] = True
```

So for example, to load the below graph, could have code:



Call createGraph(3)

Call addVertex()
Call addVertex()
Call addVertex()
Call addEdge(0, 1)
Call addEdge(1, 0)
Call addEdge(1, 2)

So maybe could also add another method such as addVertices(num) to add those vertices.

Adjacency matrix would be loaded as:

```
      0     1      2
0   False  True   False
1   True   False  True
2   False  False  False
```

If needed different labels, instead of index values, could create a Vertex class with a label and then add another array to keep the labels:

```
class Graph

    createGraph(max) where size is max size
    adj[][]    # True if exists an edge
    vertices[max]  # array of Vertex objects
    size is 0

    addVertex(label)
    create Vertex(label) object v
    vertices[size] = v
    size = size + 1

    addEdge(fromV, toV)  # still can use index for simplicity
    adj[fromV][toV] = True

class Vertex
    label
```

So using the same graph as above, if vertex had labels:
node 0 => label5
node 1 => label4
node 2 => label2

The code to load would be:
Call createGraph(3)
Call addVertex(label5)

```
Call addVertex(label4)
Call addVertex(label2)
Call addEdge(0, 1)
Call addEdge(1, 0)
Call addEdge(1, 2)
```

Adjacency matrix would be the same as before and my vertices array would have:

```
vertex-label5  vertex-label4  vertex-label2
```

If needed additional data for each vertex, that could be added as members of the Vertex class.

If needed to remove edges, that could be done with either of the above implementation such as:

```
removeEdge(fromV, toV)
    adj[fromV][toV] = False
```

To remove a vertex, that could be a bit more complicated but one way to simplify would be to have the caller need to remove each edge themselves. So we would only need to add:

```
removeVertex()
    size = size -1
```

So to remove we would code:

```
Call removeVertex()
Call removeEdge(1, 2)
```

Adjacency List

For **Adjacency List** we would implement just like any linked list but some of the operations would be a bit more complicated and again it can be implemented in many different ways depending how it will be used. May also use builtin list.

Here is one simple way to create graph that could be implemented in any programming language. In this case we do not have addVertex() because we create vertex nodes as we add edge:

```
class Vertex
    Vertex(l)
    set label=l
    set next = None   #Vertex next

class Graph

    Graph(max)
    set size=max
    vertices[max]  # for vertex objects
    addEdge(src, dst)
    create new Vertex(dst) obj
    set vertices[src].next to new Vertex obj
    set vertices[src] to new Vertex obj
```

To load the same graph as in previous examples:

```
call Graph(3)
call addEdge(0, 1)
call addEdge(1, 0)
call addEdge(1, 2)
```

So our structure would look like this:

```
0->1->None
1->0->2->None
2->None
```

Where each line gives the vertex and what other vertices it has edges to. So first line shows vertex 0 has 0,1 edge; next line has vertex 1 which has edges 1,0 and 1,2; and last line has vertex 2 has no edges.

The above output can be generated by the following code:

```
printGraph()
loop i=0 to size
    print i and “->”
    assign v = vertices[i]
    loop until v is None
        print v.label and “->”
        assign v = v.next
    print “None”
```

To remove an edge, you need to traverse that vertex’s list and find the one to delete. The code would be the same as list’s remove:

```
removeEdge(src, dst)
set v = vertices[src]
if v.label is dst  # in first element of the linked list
    if v.next is None    # there is only one element
        set vertices[src] to None
    else
        set vertices[src] to v.next
else
    loop through v until v.next is None
        if next.label is dst
            v.next = v.next.next
```

```
                    break from loop
```

To call remove:

```
Call removeEdge(1, 2)
```

Also instead of implementing the linked list within Graph as above, you could use your existing Python's list and then you just use the operations for create, add, and remove.

Traversal

To implement the DFS you can then use a recursive implementation such as below. Note that if the graph is not connected, the traversal algorithm will not reach all vertices for some starting points.

```
DFS(int n)
Initialize to False (not visited) for each visited[0..size]   # visited has bool : True/False
Call DFSProcess(n, visited)

DFSProcess(n, visited[])
Set visited[n] = True
Do some action for that vertex such as print vertex value
Loop through all neighbors of n
    If not visited[x]  # x is the neighbor vertex
        Call DFSProcess(x, visited)
```

So for example if we create graph and print the label as we process of visiting node we have:

```
Call Graph(4);
# add vertex if needed (implementation dependent)
Call addEdge(0, 2);
Call addEdge(0, 1);
Call addEdge(1, 0);
Call addEdge(2, 3);
Call addEdge(2, 3);
Call addEdge(3, 1);

Call DFS(2);
```

```
0->1->2->None
1->2->None
2->0->3->None
3->1->None
```

Output:

```
2 0 1 3
```