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

class Graph

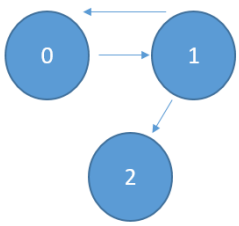
```
boolean adj[][]    // true if exists an edge
int size           // number of vertices

createGraph(max)
initialize adj to [max][max]  as needed  // depends on programming language used
size is 0

addVertex()
size = size + 1

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

So for example, to load the below graph, I 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 I could also add another method such as addVertices(num) to add that many new vertices.

My adjacency matrix would be loaded as:

```
      0     1      2
0   false  true   false
1   true   false  true
2   false  false  false
```

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

class Graph

boolean adj[][] // true if exists an edge
int size // number of vertices
Vertex vertices[]

createGraph(max) where size is max size
initialize adj to [max][max] as needed // depends on programming language used
vertices[max]
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 my vertex had labels:
node 0 => 5
node 1 => 4

node 2 => 2

The code to load would be:
Call createGraph(3)
Call addVertex(5)
Call addVertex(4)
Call addVertex(2)
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:
5 4 2

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

If I 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. Some programming languages may also have a builtin linked list ADT that could be used.

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

class Vertex
    int label
    Vertex next

    Vertex(l)

set label=l
set next = null

class Graph
int size
Vertex vertices[]

Graph(max)
set size=max
initialize all vertices[i] to null

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->null
1->0->2->null
2->null
```

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 null
        print v.label and "->"
        assign v = v.next
    print "null"

To remove an edge, you need to traverse that vertex's linked list and find the one to delete. The code would be the same as linked 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 null    // there is only one element
                set vertices[src] to null
        else
                set vertices[src] to v.next
else
        loop through v until v.next is null
                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 linked list ADT and then you just use that ADTs 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]
Call DFSProcess(n, visited)

DFSProcess(n, boolean 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->null
1->2->null
2->0->3->null
3->1->null
```

Output:

```
2 0 1 3
```