Scientific Programming Practical 18

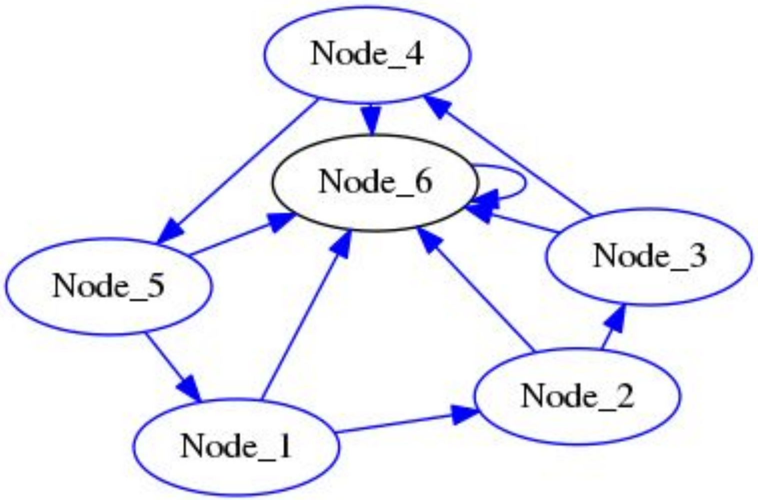
Introduction

Luca Bianco - Academic Year 2018-19
luca.bianco@fmach.it

Graphs

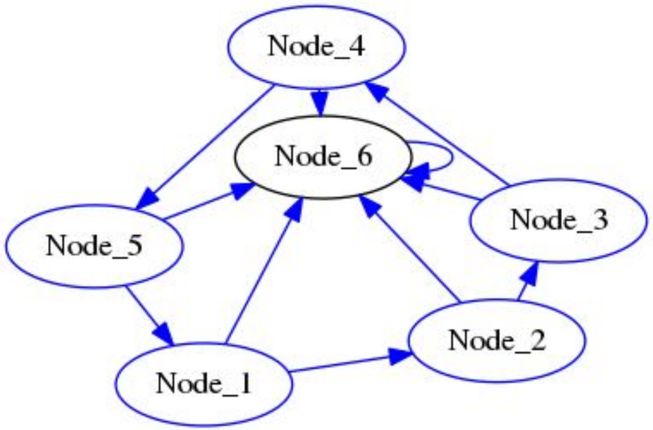
Graphs are mathematical structures made of two key elements: **nodes** (or **vertices**) and **edges**. Nodes are things that we want to represent and edges are relationships among the objects.

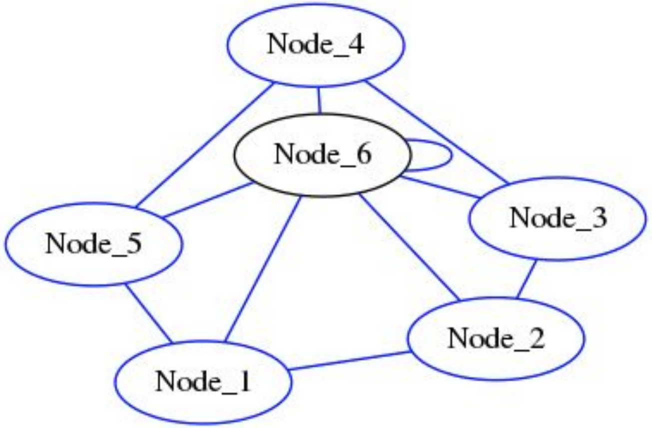
Mathematically, a graph $G=(N,E)$ where N is a set of nodes and $E=N\times N$ is the set of edges.



Transitive vs. non-transitive relations

Directed vs. undirected graphs
(arrows vs. lines)



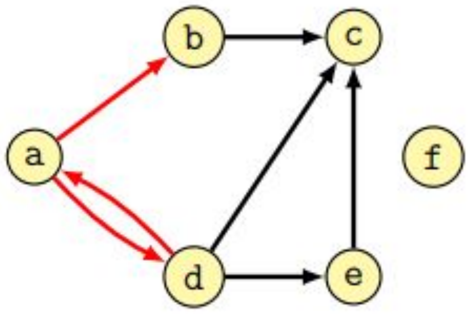


X is sibling of Y implies **Y is sibling of X** (undirected)

X is father of Y (directed)

Some terminology

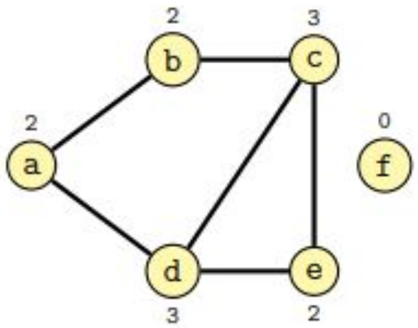
- Vertex v is adjacent to u if and only if $(u, v) \in E$.
- In an undirected graph, the adjacency relation is symmetric
- An edge (u, v) is said to be incident from u to v

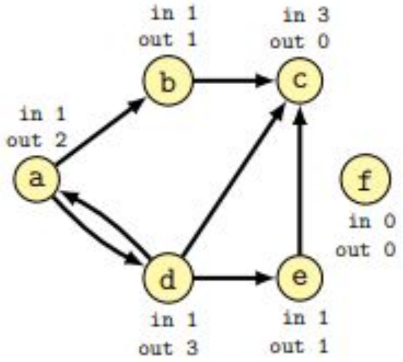


- (a, b) is incident from a to b
- (a, d) is incident from a to d
- (d, a) is incident from d to a
- b is adjacent to a
- d is adjacent to a
- a is adjacent to d

Some terminology

The **degree** of a node is the number of connections it has with other nodes. In directed graphs the **in-degree** is the number of **incoming** edges, while the **out-degree** is the number of **outgoing** edges.



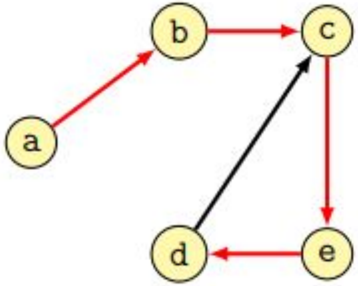


Some terminology

A **path** in the graph is a sequence of nodes connected by edges.

Path

In a graph $G = (V, E)$, a path C of length k is a sequence of nodes $u_0, u_1, \ldots, u_k$ such that $(u_i, u_{i+1} \in E)$ for $0 \leq i \leq k-1$.



Example: `a, b, c, e, d` is a path in the graph of length 4.

It is also the shortest path between `a` and `d`

Note: a path is said to be simple if all its nodes are distinct

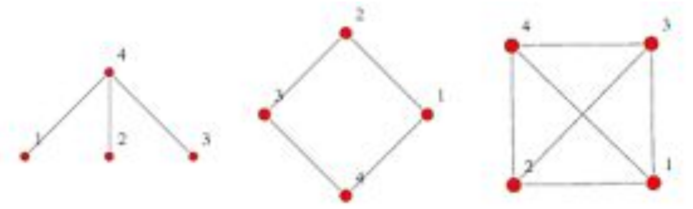
Graph: ADT

Graphs are dynamic data structures in which nodes and edges can be added/removed.

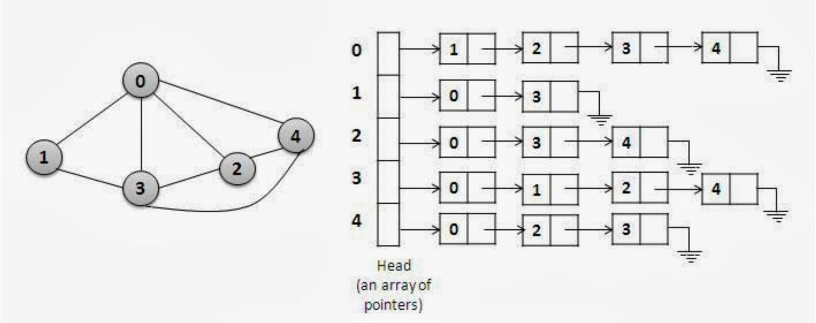
GRAPH	
Graph( )	% Create a new graph
SET size()	% Returns the number of nodes
SET V()	% Returns the set of all nodes
SET adj(NODE u)	% Returns the set of nodes adjacent to u
insertNode(NODE u)	% Add node u to the graph
insertEdge(NODE u, NODE v)	% Add edge (u, v) to the graph
deleteNode(NODE u)	% Removes node u from the graph
deleteEdge(NODE u, NODE v)	% Removes edge (u, v) from the graph

Two possible implementations

Graphs can be implemented as **adjacency matrices** or **adjacency linked lists**.



$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$



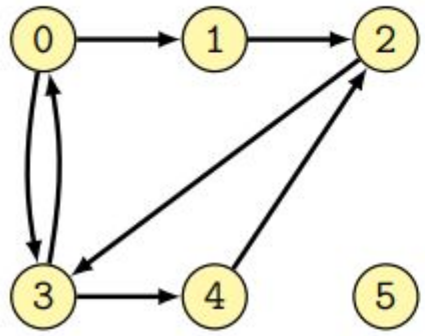
Adjacency List Representation of Graph

Adjacency matrix

A **square matrix *G*** having the size ***N×N*** where ***N* is the number of nodes**, is used to represent every possible connection among the nodes of the graph. In particular ***G*[*i*,*j*]=1** if the graph has a **edge** connecting node ***i* to node *j***, otherwise ***G*[*i*,*j*]=0**.

$$m_{uv} = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

$$\text{Space} = n^2 \text{ bits}$$

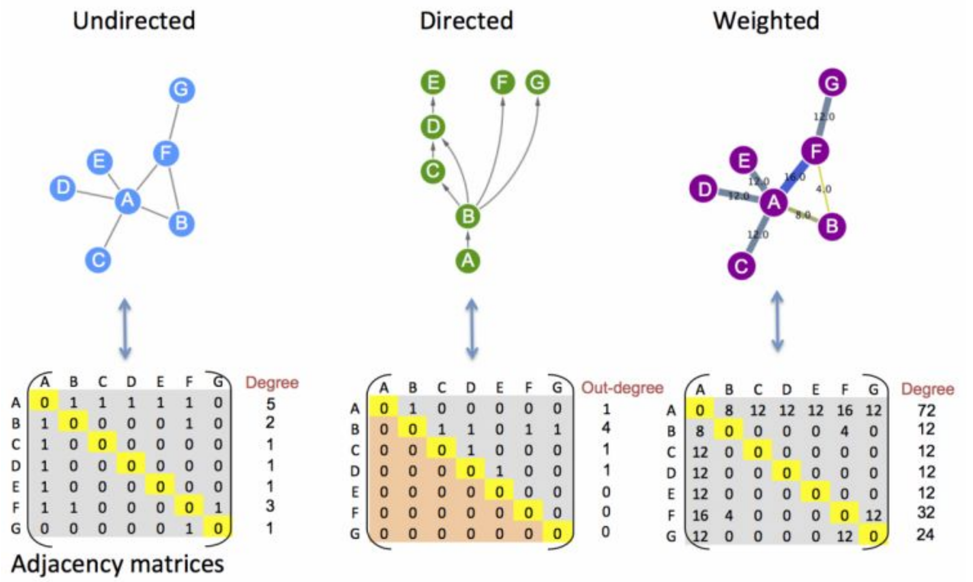


$$\begin{array}{c} \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{2} \\ \mathbf{3} \\ \mathbf{4} \\ \mathbf{5} \end{array} \begin{array}{c} \begin{array}{cccccc} \mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5} \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Adjacency matrix

This representation of a graph has some advantages and disadvantages:

- it is **quite flexible** as it is possible to put **weights** on the values of the matrix instead of only 0 and 1;
- it is quite **quick to check the presence of an edge** (both ways!): this just requires a lookup in the matrix G;
- it uses a lot of **space NxN** and **most of the values often are 0** (a lot of space is therefore wasted);
- in **undirected** graphs, the matrix is **symmetric** therefore h**alf of the space can be saved**.



Adjacency matrix: implementation

```python
class DiGraphAsAdjacencyMatrix:
    def __init__(self):
        #would be better a set, but I need an index
        self.__nodes = list()
        self.__matrix = list()

    def __len__(self):
        """gets the number of nodes"""
        return len(self.__nodes)

    def nodes(self):
        return self.__nodes

    def matrix(self):
        return self.__matrix

    def __str__(self):
        header = "\t".join([n for n in self.__nodes])
        data = ""
        for i in range(0,len(self.__matrix)):
            data += str(self.__nodes[i]) + "\t"
            data += "\t".join([str(x) for x in self.__matrix[i]]) + "\n"

        return "\t"+ header +"\n" + data
```

```python
def insertNode(self, node):
    #add the node if not there.
    if node not in self.__nodes:
        self.__nodes.append(node)
        #add a row and a column of zeros in the matrix
        if len(self.__matrix) == 0:
            #first node
            self.__matrix = [[0]]
        else:
            N = len(self.__nodes)
            for row in self.__matrix:
                row.append(0)
            self.__matrix.append([0 for x in range(N)])

def insertEdge(self, node1, node2, weight):
    i = -1
    j = -1
    if node1 in self.__nodes:
        i = self.__nodes.index(node1)
    if node2 in self.__nodes:
        j = self.__nodes.index(node2)
    if i != -1 and j != -1:
        self.__matrix[i][j] = weight
```

Adjacency matrix: implementation

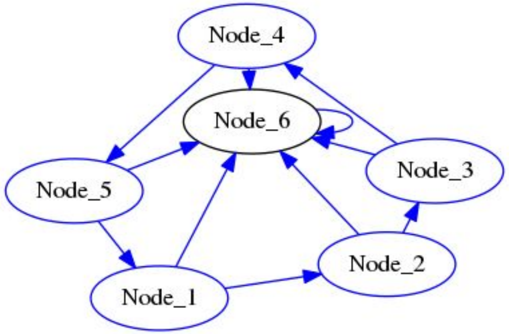
```python
class DiGraphAsAdjacencyMatrix:
    def __init__(self):
        #would be better a set, but I need an index
        self.__nodes = list()
        self.__matrix = list()

    def __len__(self):
        """gets the number of nodes"""
        return len(self.__nodes)

    def nodes(self):
        return self.__nodes

    def matrix(self):
        return self.__matrix

    def __str__(self):
        header = "\t".join([n for n in self.__nodes])
        data = ""
        for i in range(0,len(self.__matrix)):
            data += str(self.__nodes[i]) + "\t"
            data += "\t".join([str(x) for x in self.__matrix[i]]) + "\n"

        return "\t"+ header +"\n" + data
```

```python
def deleteEdge(self, node1,node2):
    """removing an edge means to set its
    corresponding place in the matrix to 0"""
    i = -1
    j = -1
    if node1 in self.__nodes:
        i = self.__nodes.index(node1)
    if node2 in self.__nodes:
        j = self.__nodes.index(node2)
    if i != -1 and j != -1:
        self.__matrix[i][j] = 0

def deleteNode(self, node):
    """removing a node means removing
    its corresponding row and column in the matrix"""
    i = -1

    if node in self.__nodes:
        i = self.__nodes.index(node)
    #print("Removing {} at index {}".format(node, i))
    if node != -1:
        self.__matrix.pop(i)
        for row in self.__matrix:
            row.pop(i)
        self.__nodes.pop(i)

def adjacent(self, node, incoming = True):
    """Your treat! (see exercise 1)"""

def edges(self):
    """Your treat! (see exercise1). Returns all the edges"""
```

Adjacency matrix: implementation

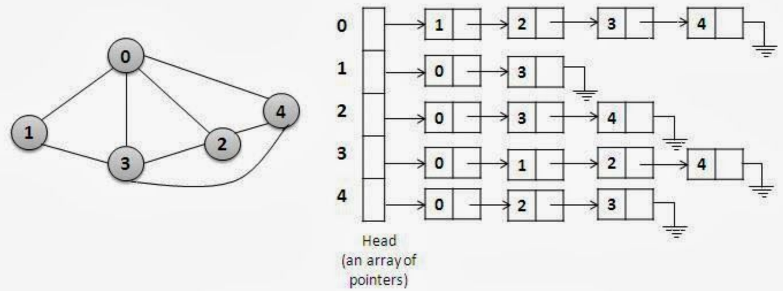
```python
if __name__ == "__main__":
    G = DiGraphAsAdjacencyMatrix()

    for i in range(6):
        n = "Node_{}".format(i+1)
        G.insertNode(n)

    for i in range(0,4):
        n = "Node_" + str(i+1)
        six = "Node_6"
        n_plus = "Node_" + str((i+2) % 6)
        G.insertEdge(n, n_plus,0.5)
        G.insertEdge(n, six,1)
    G.insertEdge("Node_5", "Node_1", 0.5)
    G.insertEdge("Node_5", "Node_6", 1)
    G.insertEdge("Node_6", "Node_6", 1)
    print(G)

    G.insertNode("Node_7")
    G.insertEdge("Node_1", "Node_7", -1)
    G.insertEdge("Node_2", "Node_7", -2)
    G.insertEdge("Node_5", "Node_7", -5)
    G.insertEdge("Node_7", "Node_2", -2)
    G.insertEdge("Node_7", "Node_3", -3)

    print("Size is: {}".format(len(G)))
    print("Nodes: {}".format(G.nodes()))
    print("\nMatrix:")
    print(G)
    G.deleteNode("Node_7")
    G.deleteEdge("Node_6", "Node_2")
    #no effect, nodes do not exist!
    G.insertEdge("72", "25",3)
    print(G)
```

```
        Node_1  Node_2  Node_3  Node_4  Node_5  Node_6
Node_1  0       0.5     0       0       0       1
Node_2  0       0       0.5     0       0       1
Node_3  0       0       0       0.5     0       1
Node_4  0       0       0       0       0.5     1
Node_5  0.5     0       0       0       0       1
Node_6  0       0       0       0       0       1

Size is: 7
Nodes: ['Node_1', 'Node_2', 'Node_3', 'Node_4', 'Node_5', 'Node_6', 'Node_7']

Matrix:
        Node_1  Node_2  Node_3  Node_4  Node_5  Node_6  Node_7
Node_1  0       0.5     0       0       0       1       -1
Node_2  0       0       0.5     0       0       1       -2
Node_3  0       0       0       0.5     0       1       0
Node_4  0       0       0       0       0.5     1       0
Node_5  0.5     0       0       0       0       1       -5
Node_6  0       0       0       0       0       1       0
Node_7  0       -2      -3      0       0       0       0

        Node_1  Node_2  Node_3  Node_4  Node_5  Node
Node_1  0       0.5     0       0       0       1
Node_2  0       0       0.5     0       0       1
Node_3  0       0       0       0.5     0       1
Node_4  0       0       0       0       0.5     1
Node_5  0.5     0       0       0       0       1
Node_6  0       0       0       0       0       1
```

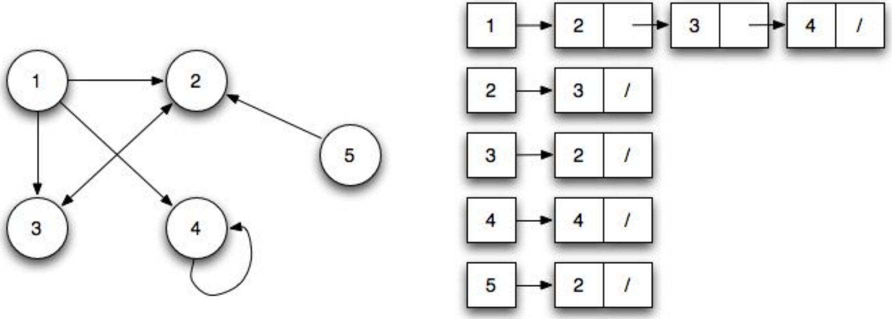


Adjacency linked list

In an **adjacency linked list** each node *N* has a linked-list of nodes connected to it in *G*. In the case of **directed graphs**, **every node contains a list of all the nodes** reachable through some **outgoing** edges, while in the case of **undirected** graphs the list will be of **all nodes connected together** by means of an edge.



Adjacency List Representation of Graph

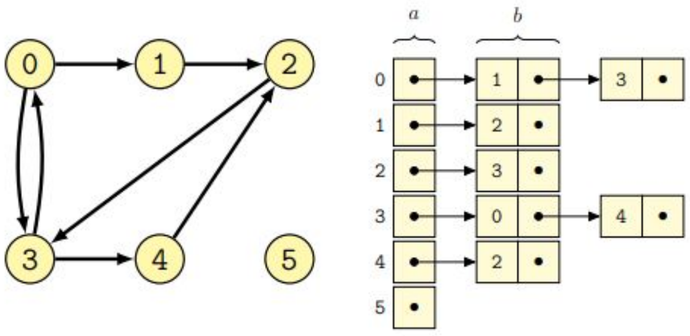


Adjacency linked list

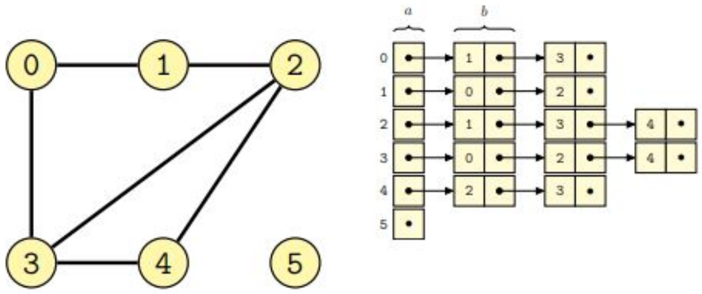
The implementation through adjacency linked lists has some advantages and disadvantages:

- it is **flexible**, **nodes can be complex objects** (with the only requirement of the **attribute linking to the neighboring nodes**);
- in general, it **uses less space**, only that required by the pointers encoding for the existing edges;
- **checking presence of an edge is in general slower** (this requires going through the list of source node);
- getting **all incoming edges of a node is slow** (requires going through all nodes!). A workaround to this problem is to **store not only outgoing-edges but also incoming edges** (but this requires more memory).

$G.\text{adj}(u) = \{v \mid (u, v) \in E\}$ Space $= an + bm$ bits



$G.\text{adj}(u) = \{v \mid (u, v) \in E\}$ Space $= an + 2 \cdot bm$

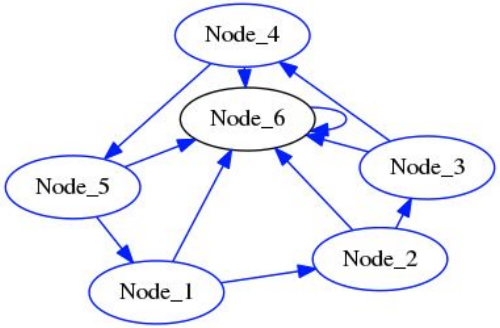


Adjacency linked list: implementation

```python
class DiGraphLL:
    def __init__(self):
        """Every node is an element in the dictionary.
        The key is the node id and the value is a dictionary
        with key second node and value the weight
        """
        self.__nodes = dict()

    def insertNode(self, node):
        test = self.__nodes.get(node, None)

        if test == None:
            self.__nodes[node] = {}
            #print("Node {} added".format(node))

    def insertEdge(self, node1, node2, weight):
        test = self.__nodes.get(node1, None)
        test1 = self.__nodes.get(node2, None)
        if test != None and test1 != None:
            #if both nodes exist othewise don't do anything
            test = self.__nodes[node1].get(node2, None)
            if test != None:
                exStr= "Edge {} --> {} already existing.".format(node1,
                                                                 node2)
                raise Exception(exStr)
            else:
                #print("Inserted {}-->{} ({})".format(node1,node2,weight))
                self.__nodes[node1][node2] = weight
```

Instead of a linked list we use a dictionary for speed

{ 'Node_5': {'Node_6': 1, 'Node_1': 0.5},
'Node_1': {'Node_6': 1, 'Node_2': 0.5},
'Node_2': {'Node_6': 1, 'Node_3': 0.5},
'Node_6': {'Node_6': 1},
'Node_3': {'Node_6': 1, 'Node_4': 0.5},
'Node_4': {'Node_6': 1, 'Node_5': 0.5} }

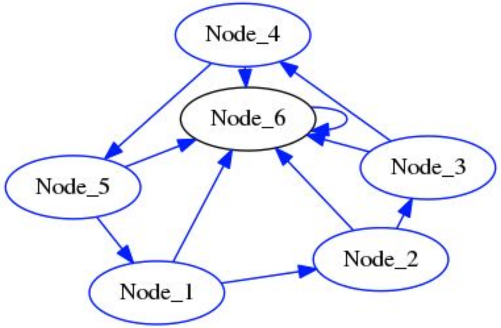


Adjacency linked list: implementation

```python
class DiGraphLL:
    def __init__(self):
        """Every node is an element in the dictionary.
        The key is the node id and the value is a dictionary
        with key second node and value the weight
        """
        self.__nodes = dict()

    def insertNode(self, node):
        test = self.__nodes.get(node, None)

        if test == None:
            self.__nodes[node] = {}
            #print("Node {} added".format(node))

    def insertEdge(self, node1, node2, weight):
        test = self.__nodes.get(node1, None)
        test1 = self.__nodes.get(node2, None)
        if test != None and test1 != None:
            #if both nodes exist othewise don't do anything
            test = self.__nodes[node1].get(node2, None)
            if test != None:
                exStr= "Edge {} --> {} already existing.".format(node1,
                                                                  node2)

                raise Exception(exStr)
            else:
                #print("Inserted {}-->{} ({})".format(node1,node2,weight))
                self.__nodes[node1][node2] = weight
```

```python
def deleteNode(self, node):
    test = self.__nodes.get(node, None)
    if test != None:
        self.__nodes.pop(node)
    # need to loop through all the nodes!!!
    for n in self.__nodes:
        test = self.__nodes[n].get(node, None)
        if test != None:
            self.__nodes[n].pop(node)

def deleteEdge(self, node1,node2):
    test = self.__nodes.get(node1, None)
    if test != None:
        test = self.__nodes[node1].get(node2, None)
        if test != None:
            self.__nodes[node1].pop(node2)

def __len__(self):
    return len(self.__nodes)

def nodes(self):
    return list(self.__nodes.keys())

def graph(self):
    return self.__nodes

def __str__(self):
    ret = ""
    for n in self.__nodes:
        for edge in self.__nodes[n]:

            ret += "{} -- {} --> {}\n".format(str(n),
                                              str(self.__nodes[n][edge]),
                                              str(edge))
    return ret
```

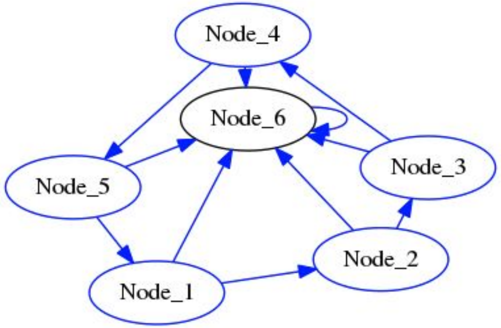
Adjacency linked list: implementation




```python
if __name__ == "__main__":
    G = DiGraphLL()
    for i in range(6):
        n = "Node_{}".format(i+1)
        G.insertNode(n)

    for i in range(0,4):
        n = "Node_" + str(i+1)
        six = "Node_6"
        n_plus = "Node_" + str((i+2) % 6)
        G.insertEdge(n, n_plus,0.5)
        G.insertEdge(n, six,1)
    G.insertEdge("Node_5", "Node_1", 0.5)
    G.insertEdge("Node_5", "Node_6", 1)
    G.insertEdge("Node_6", "Node_6", 1)
    print(G)

    G.insertNode("Node_7")
    G.insertEdge("Node_1", "Node_7", -1)
    G.insertEdge("Node_2", "Node_7", -2)
    G.insertEdge("Node_5", "Node_7", -5)
    G.insertEdge("Node_7", "Node_2", -2)
    G.insertEdge("Node_7", "Node_3", -3)

    print("Size is: {}".format(len(G)))
    print("Nodes: {}".format(G.nodes()))
    print("Graph:")
    print(G)
    G.deleteNode("Node_7")
    G.deleteEdge("Node_6", "Node_2")
    #nodes do not exist! Therefore nothing happens!
    G.insertEdge("72", "25",3)
    print(G)
    print("Nodes: {}".format(G.nodes()))
    G.deleteEdge("72","25")
    print("Nodes: {}".format(G.nodes()))
    print(G)
```

```
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_2 -- 1 --> Node_6
Node_2 -- 0.5 --> Node_3
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_6 -- 1 --> Node_6

Size is: 7
Nodes: ['Node_5', 'Node_4', 'Node_7', 'Node_1', 'Node_2', 'Node_3', 'Node_6']
Graph:
Node_5 -- -5 --> Node_7
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_7 -- -2 --> Node_2
Node_7 -- -3 --> Node_3
Node_1 -- -1 --> Node_7
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_2 -- 1 --> Node_6
Node_2 -- -2 --> Node_7
Node_2 -- 0.5 --> Node_3
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_6 -- 1 --> Node_6
```

Adjacency linked list: implementation




```python
if __name__ == "__main__":
    G = DiGraphLL()
    for i in range(6):
        n = "Node_{}".format(i+1)
        G.insertNode(n)

    for i in range(0,4):
        n = "Node_" + str(i+1)
        six = "Node_6"
        n_plus = "Node_" + str((i+2) % 6)
        G.insertEdge(n, n_plus,0.5)
        G.insertEdge(n, six,1)
    G.insertEdge("Node_5", "Node_1", 0.5)
    G.insertEdge("Node_5", "Node_6", 1)
    G.insertEdge("Node_6", "Node_6", 1)
    print(G)

    G.insertNode("Node_7")
    G.insertEdge("Node_1", "Node_7", -1)
    G.insertEdge("Node_2", "Node_7", -2)
    G.insertEdge("Node_5", "Node_7", -5)
    G.insertEdge("Node_7", "Node_2", -2)
    G.insertEdge("Node_7", "Node_3", -3)

    print("Size is: {}".format(len(G)))
    print("Nodes: {}".format(G.nodes()))
    print("Graph:")
    print(G)
    G.deleteNode("Node_7")
    G.deleteEdge("Node_6", "Node_2")
    #nodes do not exist! Therefore nothing happens!
    G.insertEdge("72", "25",3)
    print(G)
    print("Nodes: {}".format(G.nodes()))
    G.deleteEdge("72","25")
    print("Nodes: {}".format(G.nodes()))
    print(G)
```

```
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_2 -- 1 --> Node_6
Node_2 -- 0.5 --> Node_3
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_6 -- 1 --> Node_6

Size is: 7
Nodes: ['Node_5', 'Node_4', 'Node_7', 'Node_1', 'Node_2', 'Node_3', 'Node_6']
Graph:
Node_5 -- -5 --> Node_7
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_7 -- -2 --> Node_2
Node_7 -- -3 --> Node_3
Node_1 -- -1 --> Node_7
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_2 -- 1 --> Node_6
Node_2 -- -2 --> Node_7
Node_2 -- 0.5 --> Node_3
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_6 -- 1 --> Node_6
```

```
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_2 -- 0.5 --> Node_3
Node_2 -- 1 --> Node_6
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_6 -- 1 --> Node_6

Nodes: ['Node_5', 'Node_4', 'Node_3', 'Node_2', 'Node_1', 'Node_6']
Nodes: ['Node_5', 'Node_4', 'Node_3', 'Node_2', 'Node_1', 'Node_6']
Node_5 -- 0.5 --> Node_1
Node_5 -- 1 --> Node_6
Node_4 -- 0.5 --> Node_5
Node_4 -- 1 --> Node_6
Node_3 -- 0.5 --> Node_4
Node_3 -- 1 --> Node_6
Node_2 -- 0.5 --> Node_3
Node_2 -- 1 --> Node_6
Node_1 -- 0.5 --> Node_2
Node_1 -- 1 --> Node_6
Node_6 -- 1 --> Node_6
```

http://qcbprolab.readthedocs.io/en/latest/practical18.html

Exercises

1. Consider the Graph class `DiGraphAsAdjacencyMatrix`. Add the following methods:

- `adjacent(self, node)` : given a node returns all the nodes connected to it (both incoming and outgoing);
- `adjacentEdge(self, node, incoming=True)` : given a node, returns all the nodes close to it (incoming if "incoming=True" or outgoing if "incoming = False") as a list of pairs (node, other, weight);
- `edges(self)` : returns all the edges in the graph as pairs (i,j, weight);
- `edgeIn(self, node1, node2)` : check if the edge node1 -> node2 is in the graph;

You can download the code written above to extend it from here: pract18_ex1.py

Test the code with:

```
G = DiGraphAsAdjacencyMatrix()
for i in range(6):
    n = "Node_{}".format(i+1)
    G.insertNode(n)

for i in range(0,4):
    n = "Node_" + str(i+1)
    six = "Node_6"
    n_plus = "Node_" + str((i+2) % 6)
    G.insertEdge(n, n_plus,0.5)
    G.insertEdge(n, six,1)
G.insertEdge("Node_5", "Node_1", 0.5)
G.insertEdge("Node_5", "Node_6", 1)
G.insertEdge("Node_6", "Node_6", 1)


G.insertNode("Node_7")
G.insertEdge("Node_1", "Node_7", -1)
G.insertEdge("Node_2", "Node_7", -2)
G.insertEdge("Node_5", "Node_7", -5)
G.insertEdge("Node_7", "Node_2", -2)
G.insertEdge("Node_7", "Node_3", -3)
```