**Sepcification:**

def \_\_init\_\_(self, file\_name = None):

the constructor of the graph class

parameter – the name of the file from which the

graph gets its data

def \_\_loadGraph(self):

(private method) load a graf from the file

def \_\_saveGraph(self):

(private method) save a graph to the file

def getNumberOfNodes(self):

return the number of nodes of the graph

def parseNodes(self):

iterate through all the vertices

each call returns a vertex

def checkEdge(self, source, destination):

check if a specified edge exists, returns true/false

parameters – source and destination of the edge

def getEdgeWeight(self, source, destination):

get the weight of an edge

return – (int) the weight of the given edge

def setEdgeWeight(self, source, destination, new\_value):

set the weight of an edge

def getOutDegree(self, node):

return how many outbound edges has the vertex

def getInDegree(self, node):

return how many inblund edges has the vertex

def parseEdgesOut(self, source):

for a given vertex iterate through all the outbound vertices. each call returns a vertex

def parseEdgesIn(self, destination):

for a given vertex iterate through all the inbound vertices. each call returns a vertex

def addNode(self, new\_node):

add a new vertex

def addEdge(self, source, destination, weight):

add a new edge

def removeNode(self, node):

remove a vertex

def removeEdge(self, source, destination):

remove an edge

def copy(self):

create a deep copy of the whole graph

**Implemetation:**

import copy

import random

from random import randint

class NodeNotFound(Exception):

pass

class NodeAlreadyExists(Exception):

pass

class EdgeNotFound(Exception):

pass

class EdgeAlreadyExists(Exception):

pass

class IntegerError(Exception):

pass

class Graph(object)

def \_\_init\_\_(self, file\_name = None):

self.\_\_file\_name = file\_name

self.\_\_number\_of\_nodes = 0

self.\_\_number\_of\_edges = 0

self.\_\_nodes = []

self.\_\_outbound\_edges = {}

self.\_\_inbound\_edges = {}

self.\_\_edges\_weights = {}

if self.\_\_file\_name != None:

self.\_\_loadGraph()

def \_\_loadGraph(self):

file = open(self.\_\_file\_name, *"r"*)

current\_line = file.readline().strip()

parameters = current\_line.split(*" "*)

self.\_\_number\_of\_nodes = int(parameters[0])

self.\_\_number\_of\_edges = int(parameters[1])

for i in range(0, self.\_\_number\_of\_nodes):

self.\_\_nodes.append(i)

self.\_\_outbound\_edges[i] = []

self.\_\_inbound\_edges[i] = []

for i in range(0, self.\_\_number\_of\_edges):

current\_line = file.readline().strip()

parameters = current\_line.split(*" "*)

source = int(parameters[0])

destination = int(parameters[1])

cost = int(parameters[2])

self.\_\_outbound\_edges[source].append(destination)

self.\_\_inbound\_edges[destination].append(source)

self.\_\_edges\_weights[(source, destination)] = cost

file.close()

def \_\_saveGraph(self):

file = open(self.\_\_file\_name)

line = str(self.\_\_number\_of\_nodes) + *" "* + str(self.\_\_number\_of\_edges) + *"\n"*

file.write(line)

for edge in self.\_\_edges\_weights.keys():

line = str(edge[0]) + *" "* + str(edge[1]) + *" "* + str(self.\_\_edges\_weights[edge]) + *"\n"*

file.write(line)

def getNumberOfNodes(self):

return self.\_\_number\_of\_nodes

def parseNodes(self):

for node in self.\_\_nodes:

yield node

def checkEdge(self, source, destination):

try:

source = int(source)

destination = int(destination)

except TypeError:

return False

return destination in self.\_\_outbound\_edges[source]

def getEdgeWeight(self, source, destination):

if not self.checkEdge(source, destination):

raise EdgeNotFound

source = int(source)

destination = int(destination)

return self.\_\_edges\_weights[(source, destination)]

def setEdgeWeight(self, source, destination, new\_value):

if not self.checkEdge(source, destination):

raise EdgeNotFound

source = int(source)

destination = int(destination)

try:

new\_value = int(new\_value)

except TypeError:

raise IntegerError

self.\_\_edges\_weights[(source, destination)] = new\_value

def getOutDegree(self, node):

try:

node = int(node)

except TypeError:

return False

return len(self.\_\_outbound\_edges[node])

def getInDegree(self, node):

try:

node = int(node)

except TypeError:

return False

return len(self.\_\_inbound\_edges[node])

def parseEdgesOut(self, source):

try:

source = int(source)

except TypeError:

raise IntegerError

for node in self.\_\_outbound\_edges[source]:

yield node

def parseEdgesIn(self, destination):

try:

destination = int(destination)

except TypeError:

raise IntegerError

for node in self.\_\_inbound\_edges[destination]:

yield node

def addNode(self, new\_node):

if new\_node in self.\_\_nodes:

raise NodeAlreadyExists

self.\_\_nodes.append(new\_node)

self.\_\_outbound\_edges[new\_node] = []

self.\_\_inbound\_edges[new\_node] = []

def addEdge(self, source, destination, weight):

try:

source = int(source)

destination = int(destination)

weight = int(weight)

except TypeError:

raise IntegerError

if not (source in self.\_\_nodes or destination in self.\_\_nodes):

raise NodeNotFound

if (source, destination) in self.\_\_edges\_weights.keys():

raise EdgeAlreadyExists

self.\_\_outbound\_edges[source].append(destination)

self.\_\_inbound\_edges[destination].append(source)

self.\_\_edges\_weights[(source, destination)] = weight

def removeNode(self, node):

edges\_to\_remove = []

try:

node = int(node)

except TypeError:

raise IntegerError

for destination in self.\_\_outbound\_edges[node]:

edges\_to\_remove.append((node, destination))

for source in self.\_\_inbound\_edges[node]:

edges\_to\_remove.append((source, node))

for edge in edges\_to\_remove:

self.removeEdge(edge[0], edge[1])

self.\_\_nodes.remove(node)

def removeEdge(self, source, destination):

try:

source = int(source)

destination = int(destination)

except TypeError:

raise IntegerError

self.\_\_outbound\_edges[source].remove(destination)

self.\_\_inbound\_edges[destination].remove(source)

self.\_\_edges\_weights.pop((source, destination))

def copy(self):

new\_graph = copy.deepcopy(self)

return new\_graph