




Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

	<b>Carátula para entrega de prácticas</b>
Facultad de Ingeniería	Laboratorio de docencia

# Laboratorios de computación salas A y B

<i>Profesor</i>	Jesus Cruz Navarro
<i>Asignatura</i>	Estructura de Datos y Algoritmos II
<i>Grupo</i>	01
	06
<i>No de Práctica(s):</i>	Diego Santiago Gutierrez
<i>Integrante(s):</i>	
<i>No. de Equipo de cómputo</i>	
<i>No. de Lista o Brianda:</i>	
<i>Semestre</i>	Tercer Semestre
	14/11/2020
<i>Fecha de entrega:</i>	
<i>Observaciones</i>	

**CALIFICACIÓN:** \_\_\_\_\_



Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

## INTRODUCCIÓN:

- Diseñar e implementar las clases **Vértice** y **Grafo**, con los métodos **AgregarVertice** y **AgregarArista**, como los vistos en clase (usando como parámetros los nombres de los vértices, en lugar de pasar un objeto de tipo **Vértice**, como la práctica de la coordinación).

Se debe validar que:

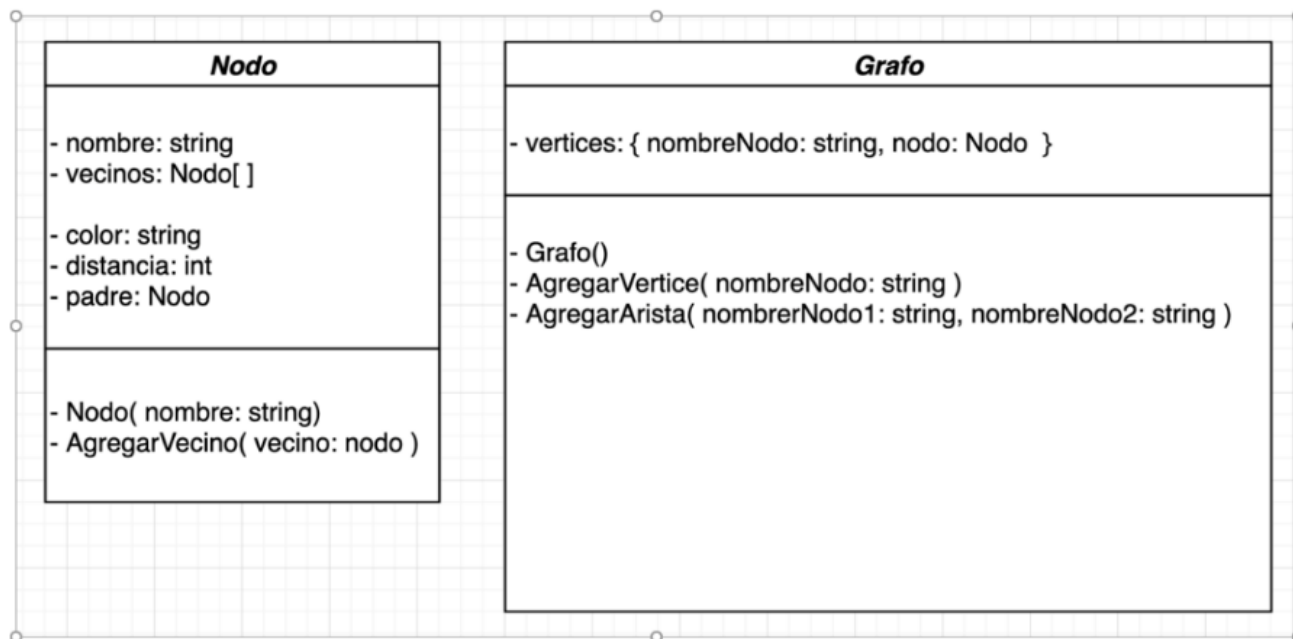
- En la clase **Vértice**, al agregar vecino, no exista ya un vecino con ese nombre de vértice.
- En la clase **Grafo**, al agregar vértice, no exista ya un vértice con ese nombre.
- En la clase **Grafo**, al agregar arista, que existan los vértices entre la arista.

En caso de error, se debe imprimir un mensaje de error en consola.

Los vértices se deben guardar en un diccionario y acceder a ellos utilizando su llave, no iterando sobre estos.

Además, ambas clases deben sobrescribir los métodos `__str__` y `__repr__` para poder imprimir el grafo desde la función `print()`. Para el caso de la clase **grafo** al imprimir se debe imprimir algo similar a una lista de adyacencia.

## Diagrama de Clases:





Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

**DESARROLLO:**

1)

```
✓ class Vertex:
    def __init__(self, name):
        #Atributos de la clase Vertex
        self.name=name
        self.neighbors = []
        self.parentNode = None
        self.color = ''
        self.distance = 0
        #Creamos metodo que añade un nuevo vecino
    def addNeighbor(self, vertex):
        #Si ya existe vertex en vecinos, indica posicion
        if vertex in self.neighbors:
            print('Vecino ya existe', vertex.name , "En el vertice: " , self.name )
            return
        #Si no existe se incerta en vecinos el vertice
        self.neighbors.append(vertex)
    def __str__(self):
        return self.name
    def __repr__(self):
        return self.name
```

**Clase Vertex:**

La clase Vertex servirá para crear los vecinos que tengan en los vertices creados, que por medio de comparaciones terminará agregando o no a un vecino. Esta clase tiene sus atributos que ayudaran a generar objetos que se puedan manipular con los mismos.



Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

```
def __init__(self, name):  
    #Atributos de la clase Ver  
    self.name=name  
    self.neighbors = []  
    self.parentNode = None  
    self.color = ''  
    self.distance = 0  
    #Creamos metodo que añade un n
```

### Atributos de la clase

Recibe como parametro el nombre. Podemos encontrar como atributos; name, neighbors, parentNode, color y distance, que son las propiedades que pueden tener los grafos.

```
def addNeighbor(self, vertex):  
    #Si ya existe vertex en vecinos, indica posicion  
    if vertex in self.neighbors:  
        print('Vecino ya existe', vertex.name , "En el vertice: " , self.name )  
        return  
    #Si no existe se incerta en vecinos el vertice  
    self.neighbors.append(vertex)
```

### Metodo addNeighbor

Recibe como parametro el vertex de la clase. Si el vertice que buscamos en el atributo neighbor está, de ser así se imprime el nombre del vertice. Si no se encuentra, en vecinos se incertará el vertice.

```
def __str__(self):  
    return self.name  
  
def __repr__(self):  
    return self.name
```

### Constructores

El método especial `__str__` que tiene por objetivo retornar un string que nos haga más legible lo que representa dicho objeto.



Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

```
class Grafo:
    #Atributos de la clase
    def __init__(self):
        self.vertices = {}

    def __str__(self):
        cadena = ''
        for nameVertex in self.vertices:
            cadena = cadena+nameVertex + "->" + str(self.vertices[nameVertex].neighbors)+"\n"
        return cadena

    def __repr__(self):
        return self.vertices

    #Metodo que permite la adición de un vertice
    def addVertex(self, nameVertex):
        #Verifica si el vertice ya existe en el diccionario de vertices
        if nameVertex in self.vertices:
            print('Ya existe: ', nameVertex)
            return False
        #Si no se cumple el if, añade el vertice en el objeto
        vertex = Vertex(nameVertex)
        self.vertices[nameVertex] = vertex
        return True

    #Metodo que añade arista
    def addBranch(self, nameVertex1, nameVertex2):
        #Si el vertice uno no está en vertices
        if not nameVertex1 in self.vertices:
            print('Error al agregar arista. No existe el vertice 1', nameVertex1)
            return False
        #Si el vertice dos no se encuentra el nombre del vertice2 en vertex
        if not nameVertex2 in self.vertices:
            print('Error al agregar arista. No existe el vertice 2', nameVertex2)
            return False

        vertex1 = self.vertices[nameVertex2]
        vertex2 = self.vertices[nameVertex1]

        vertex1.addNeighbor(vertex2)
        vertex2.addNeighbor(vertex1)
```

### Clase Grafo

La clase grafo tendrá dos metodos que servirán para agregar nuevos vertices y aristas, creando una relacion entre estos dos para finalmente llamar a la funcion vecino.



Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

```
def __init__(self):  
    self.vertices = {}
```

### Atributos de la clase

Una diccionario en donde almacenaremos los vertices llamado vetex.

```
def __str__(self):  
    cadena = ''  
    for nameVertex in self.vertices:  
        cadena = cadena+nameVertex + "->" + str(self.vertices[nameVertex].neighbors)+"\n"  
    return cadena  
  
def __repr__(self):  
    return self.vertices
```

### Constructores

El metodo str nos devuelve una cadena de impresión, para ello se irá iterando sobre Vertex en donde buscaremos en el nombre del mismo con la funcion vecinos. Al final devolvemos esta cadena.

```
def addVertex(self, nameVertex):  
    #Verifica si el vertice ya existe en el diccionario de vertices  
    if nameVertex in self.vertices:  
        print('Ya existe: ', nameVertex)  
        return False  
    #Si no se cumple el if, añade el vertice en el objeto  
    vertex = Vertex(nameVertex)  
    self.vertices[nameVertex] = vertex  
    return True
```

### Método addVertex

Recibe como parametro el nombre del vertice y realiza una comparacion, si el nombre del vertice ya se encuentra en el arreglo es imposible volver a agregarlo, regresando un falso. De lo contrario creamos una variable tipo vertice y le pasamos el nombre del vertice como parametro, volviendolo un objeto de la clase Vertice.

Despues de eso, del atributo de la clase en el indice del nombre será igual al objeto tipo vertice que hemos creado. Esto con el fin de añadir el vertice en la posicion correcta.



Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

```
def addBranch(self, nameVertex1, nameVertex2):  
    #Si el vertice uno no está en vertexs  
    if not nameVertex1 in self.vertexs:  
        print('Error al agregar arista. No existe le vertice 1', nameVertex1)  
        return False  
    #Si el vertice dos no se encuentra el nombre del vertice2 en vertex  
    if not nameVertex2 in self.vertexs:  
        print('Error al agregar arista. No existe el vertice 2', nameVertex2)  
        return False  
    #Basicamente no puedes agregar algo que no existe  
    #añadimos estos dos nuevos vertices  
    vertex1 = self.vertexs[nameVertex2]  
    vertex2 = self.vertexs[nameVertex1]  
  
    vertex1.addNeighbor(vertex2)  
    vertex2.addNeighbor(vertex1)  
  
    #¿Que modificaciones hay que hacer si es un grafo dirigido  
  
    return True
```

### Método agregar arista

Este metodo recibe como parametro el nombre del vertice uno (nameVertex1) y el nombre del vertice 2 (nameVertex2). Realiza una comparacion dentro de vertex y si no se encuentra nameVertex1 o namevertex2, entonces no es posible agregarlos y ser considerados como vecinos.



Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

2)

```
def EjemploBasico():  
  
    g = Grafo()  
  
    g.addVertex("0")  
    g.addVertex("1")  
    g.addVertex("2")  
    g.addVertex('3')  
    g.addVertex('4')  
    g.addVertex('5')  
    g.addVertex('6')  
    g.addVertex('7')  
  
    #Que pasa si se agreaga un ariste en  
    if True:  
        |  
        g.addBranch('0', 'asdasdasd')  
  
    g.addBranch('0', '1')  
    g.addBranch('0', '2')  
    g.addBranch('0', '3')  
  
    #Que pasa si se agrega un arista que  
    if True:  
        |  
        g.addBranch('1', '0')  
  
    g.addBranch('1', '2')  
    g.addBranch('2', '3')  
    g.addBranch('3', '4')  
    g.addBranch('4', '5')  
    g.addBranch('4', '6')  
    g.addBranch('5', '6')  
  
    print('###Grafo###')  
    print(g)  
  
    actualNode = g.vertices["0"];  
    while( actualNode != None ):  
        print(actualNode.name , '->')  
        actualNode = actualNode.parentNode  
  
if __name__ == "__main__":  
    EjemploBasico()
```

### Ejemplo basico

Ejemplo basico sera nuestro metodo de prueba para asegurarnos de que se este ingresando correctamente los metodos que estamos solicitando. Es por ello que primero es importante añadir el vertice con su nombre correspondiente, despues de eso agregamos los vertices para crear el ejemplo del maestro.



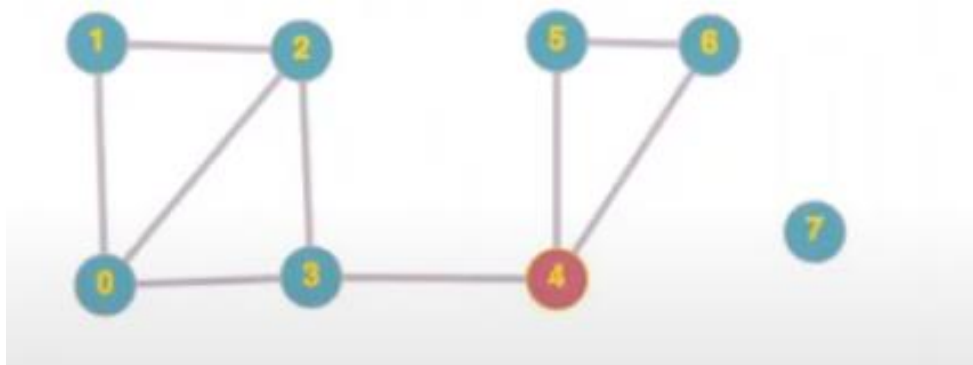


Práctica de Estudio 6: **Grafos**  
Estructura de Datos y Algoritmos Grupo 1  
Facultad de Ingeniería  
Departamento de Computación

**Salidas de ejecución**

```
Error al agregar arista. No existe el vertice 2 asdasdasd
Vecino ya existe 1 En el vertice: 0
Vecino ya existe 0 En el vertice: 1
####Grafo####
0->[1, 2, 3]
1->[0, 2]
2->[0, 1, 3]
3->[0, 2, 4]
4->[3, 5, 6]
5->[4, 6]
6->[4, 5]
7->[]
```

**Representacion visual del grafo:**



**CONCLUSIONES:**

Un es un tipo de dato abstracto que relaciona a un conjunto de nodos y conuntos aristas que establece la relacion entre estos dos. Los grafos son estructuras de datos no lineales que tienen una naturaleza generalmente dinámica, u estudio podría dividirse en dos grandes bloques; grafos dirigidos y grafos no dirigidos (pueden ser considerados un caso particular de los anteriores).

Los grafos han sido explotados ya que nos permiten entender la profundidad de las relaciones que existen entre los datos, es por ello que el comprendimiento de los mismos es vital en la computación.

Como es posible observar, el grafo que realizamos es dinamico al espacio que nosotros le asignemos, ya que este tiene la capacidad de almacenar tantos datos como sean solicitados por el usuario, el problema de hacer esto es que se vuelve dificil buscar, añadir o retirar elementos del mismo. Nosotros utilizamos vecinos para poder encontrar los grafos directamente relacionados, es decir que esten unidos por una arista.