Programación Avanzada
IIC2233 2023-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattan

Semana 15 - Estructuras Nodales

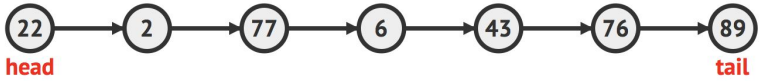
Anuncios

Jueves 23 de Noviembre 2023

1. Última actividad del semestre
2. El lunes se entrega la T3 a las 20:00, luego corren los 2 dias de entregas atrasadas.
3. La otra semana será el cierre del curso.

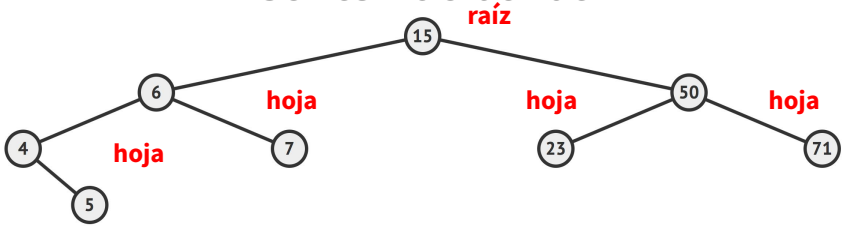
Estructuras nodales

Presentan un orden para navegar



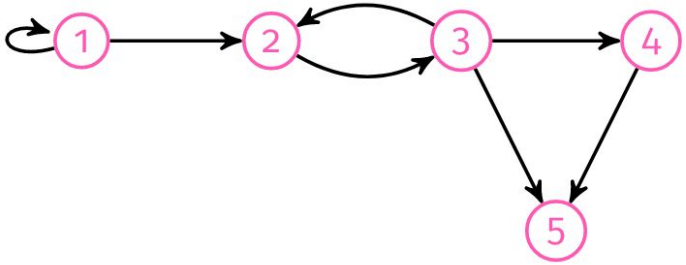

```
self.siguiente = ...
```

Contenido *bonus*




```
self.padre = ...
self.hijos = [...]
```

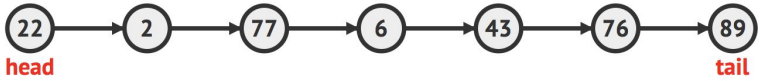
No presentan un orden para navegar




```
self.vecinos = [...]
```

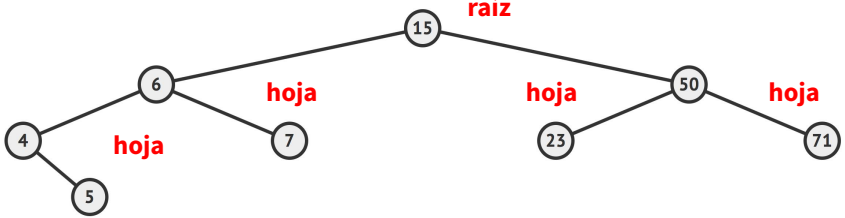
Estructuras nodales

Presentan un orden para navegar




```
self.siguiente = ...
```

Contenido *bonus*




```
self.padre = ...
self.hijos = [...]
```

No presentan un orden para navegar


```
self.vecinos = [...]
```

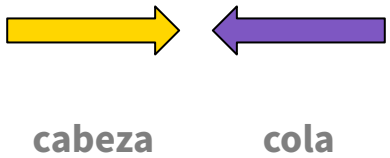
Listas Ligadas

Nodo

```
class Nodo:
    def __init__(self, valor=None):
        self.valor = valor
        self.siguiente = None
```

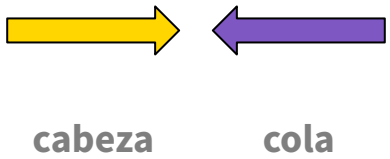
Listas Ligadas

```
class ListaLigada:
    def __init__(self):
        self.cabeza = None
        self.cola = None
```

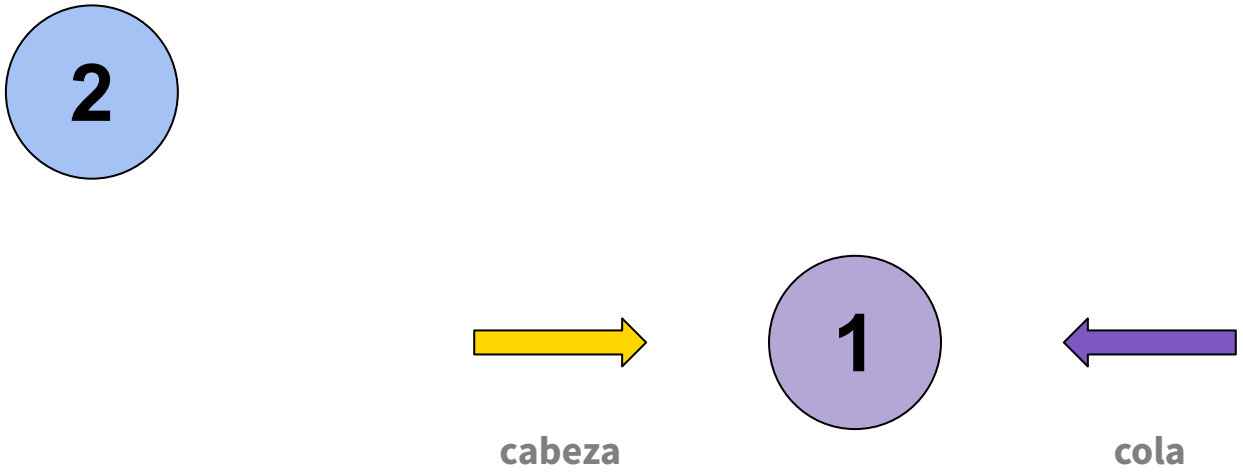


Listas Ligadas: Agregar nodos

1

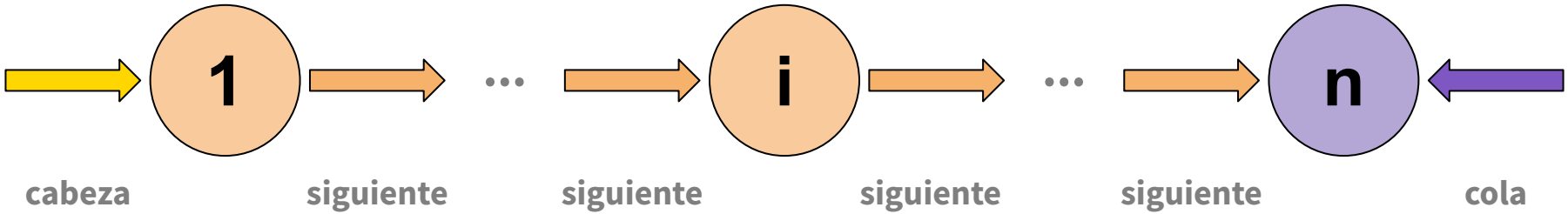


Listas Ligadas: Agregar nodos



Listas Ligadas: Agregar nodos

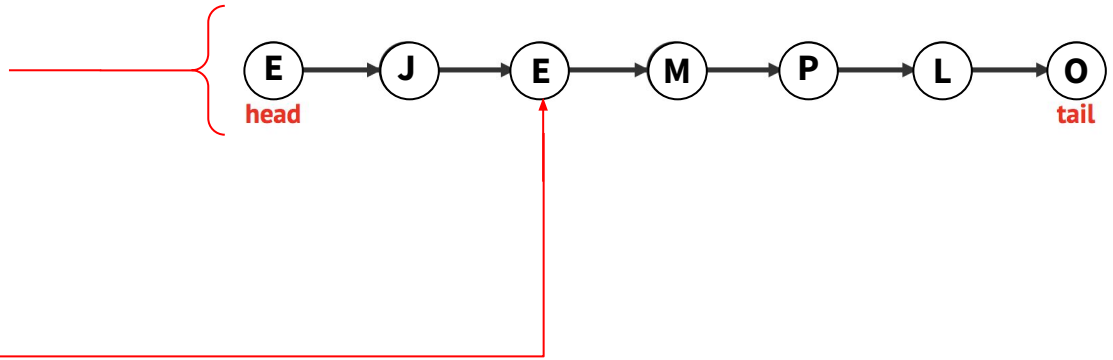
Listas Ligadas: Obtener nodo i



Cadenas de texto aleatorias

Modelamos una cadena de texto (*string*) como una lista ligada de caracteres:

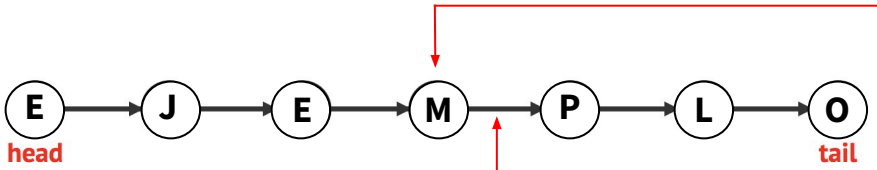
```
class TextoAleatorio
```



Cada nodo es un carácter:

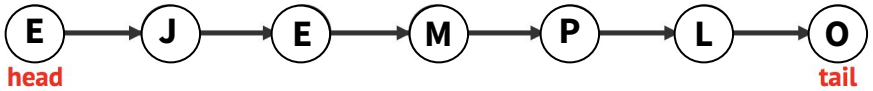
```
class Caracter
```

Cadenas de texto aleatorias




```
class Caracter:

    def __init__(self, valor):
        self.valor = valor
        self.siguiente = None
```

Cadenas de texto aleatorias




```
>> texto = TextoAleatorio("EJEMPLO")

>> print(texto)
EJEMPLO

>> texto.recorrer_texto()
En posición 0, el caracter es E
En posición 1, el caracter es J
En posición 2, el caracter es E
En posición 3, el caracter es M
...
```

```
class TextoAleatorio:
    def __init__(self, palabra):
        # Completar:
        pass

    def __repr__(self):
        # Completar:
        texto = ""
        # Concatenar caracteres
        return repr(texto)

    def recorrer_texto(self):
        # Completar:
        # Imprimir cada elemento
        pass
```

Estructuras nodales

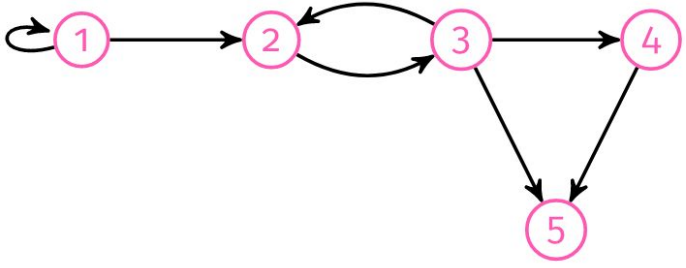
Presentan un orden para navegar


```
self.siguiente = ...
```

Contenido *bonus*


```
self.padre = ...
self.hijos = [...]
```

No presentan un orden para navegar



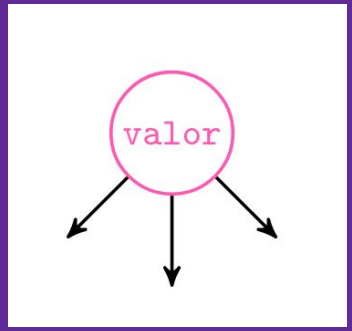

```
self.vecinos = [...]
```

Representación de grafos

- Representación con nodos
- Lista de adyacencia
- Matriz de adyacencia

Representación con nodos

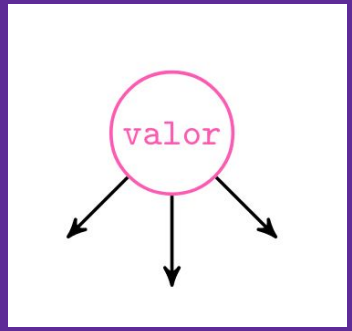
```python
class Nodo:

  def __init__(self, valor=None):
    self.valor = valor
    self.vecinos = list()

  def agregar_vecino(self, vecino):
    self.vecinos.append(vecino)
```




```python
nodo_1 = Nodo(1)
nodo_2 = Nodo(2)
nodo_3 = Nodo(3)
nodo_1.agregar_vecino(nodo_2)
```

Representación con nodos

```
class Nodo:

  def __init__(self, valor=None):
    self.valor = valor
    self.vecinos = list()

  def agregar_vecino(self, vecino):
    self.vecinos.append(vecino)
```



¿Cómo podemos representar el grafo completo en un solo objeto?

Representación con nodos

```
class Grafo:

  def __init__(self):
    self.nodos = list()
```

Representación con nodos

```
class Grafo:

  def __init__(self):
    self.nodos = list()
```

¿Cómo podemos representar el grafo completo en un solo objeto?

Lista de adyacencia

```
grafo = {
  1: [2, 3],
  2: [4, 5],
  3: [],
  4: [5],
  5: [4],
}

nodos = list(grafo.keys())
```

Lista de adyacencia

```
class Grafo:
  def __init__(self):
    self.listas_de_adyacencia = dict()

  def agregar_nodo(self, valor):
    if valor not in self.listas_de_adyacencia:
      self.listas_de_adyacencia[valor] = list()

  def agregar_conexion_dirigida(self, valor_1, valor_2):
    self.agregar_nodo(valor_1)
    self.agregar_nodo(valor_2)
    self.listas_de_adyacencia[valor_1].append(valor_2)
```

Lista de adyacencia

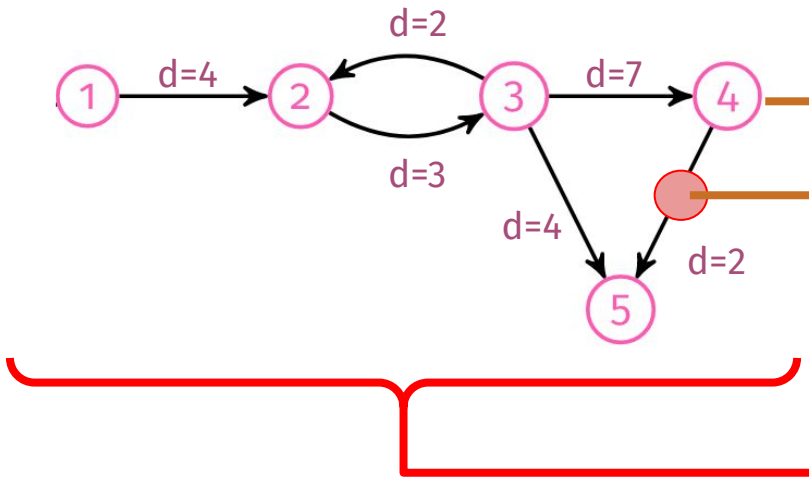
```
class Grafo:
  def __init__(self):
    self.listas_de_adyacencia = dict()

  def agregar_nodo(self, valor):
    if valor not in self.listas_de_adyacencia:
      self.listas_de_adyacencia[valor] = list()

  def agregar_conexion_dirigida(self, valor_1, valor_2):
    self.agregar_nodo(valor_1)
    self.agregar_nodo(valor_2)
    self.listas_de_adyacencia[valor_1].append(valor_2)
```

¿Existe otra forma aparte de la lista de adyacencia?

Matriz de adyacencia

```
grafo = [
  [0, 1, 1, 0, 0],
  [0, 0, 0, 1, 1],
  [0, 0, 0, 0, 0],
  [0, 0, 0, 0, 1],
  [1, 0, 0, 1, 0]
]
```

Representación basada en nodos con pesos

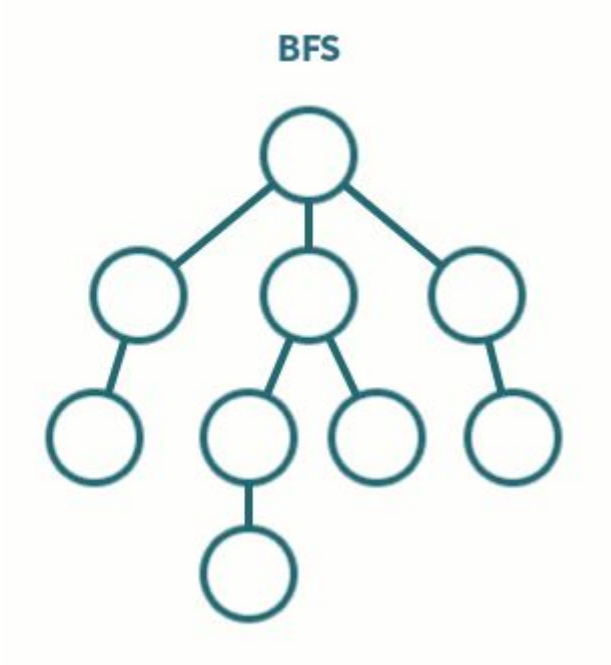



```
class Lugar:

  def __repr__(self, nombre):
    self.nombre = nombre
    self.calles = list()

  def agregar_calle(self, distancia, destino):
    self.calles.append((distancia, destino))


class Ciudad:

  def __repr__(self):
    self.lugares = list()
```

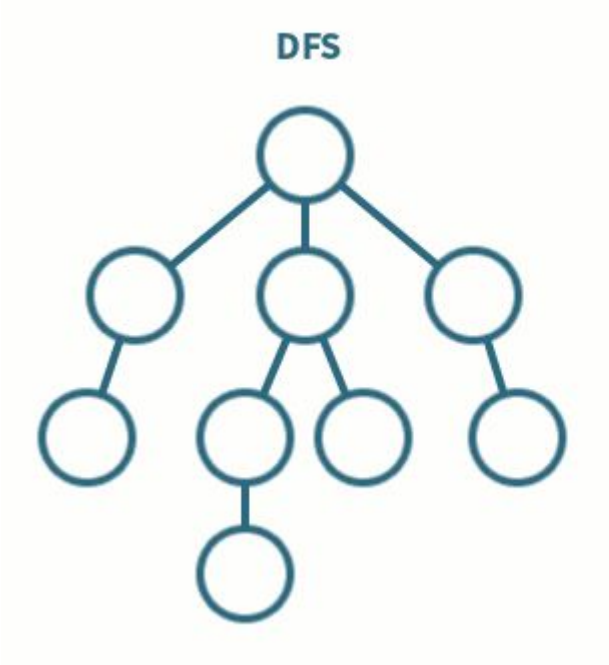
Recorrido de grafos

- BFS, búsqueda en amplitud
- DFS, búsqueda en profundidad

BFS vs DFS (en árboles)

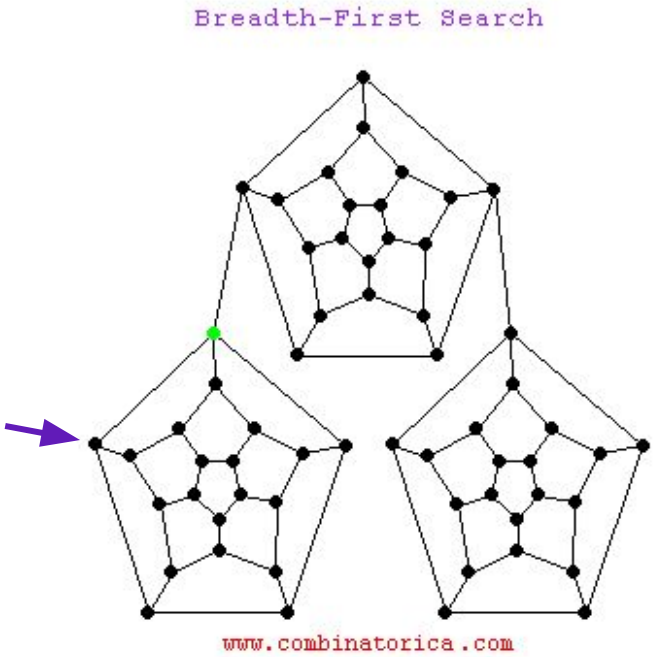


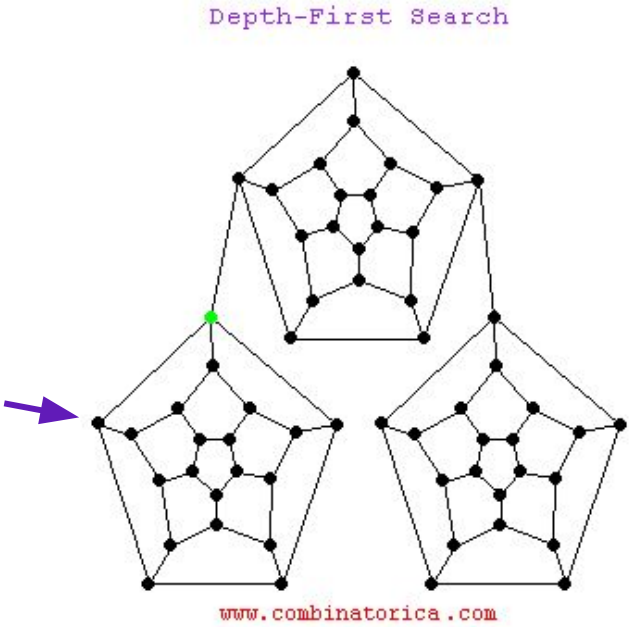
Búsqueda por amplitud



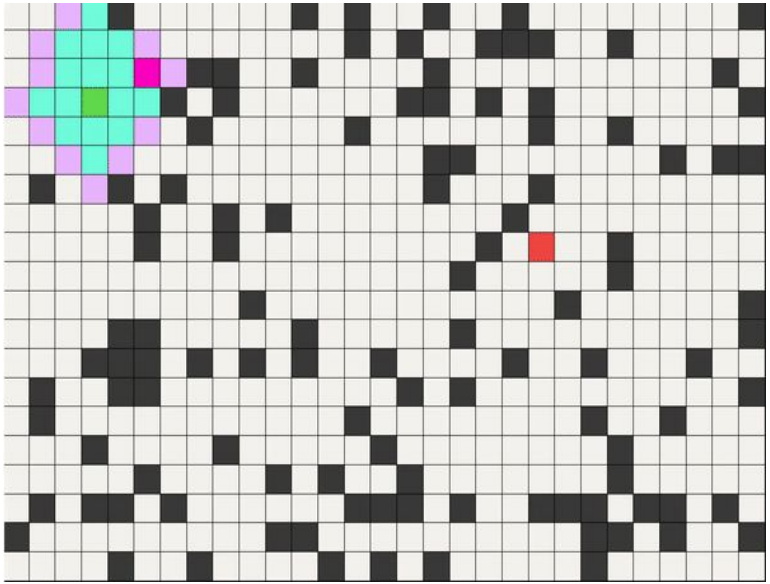
Búsqueda por profundidad

BFS vs DFS (en grafos)

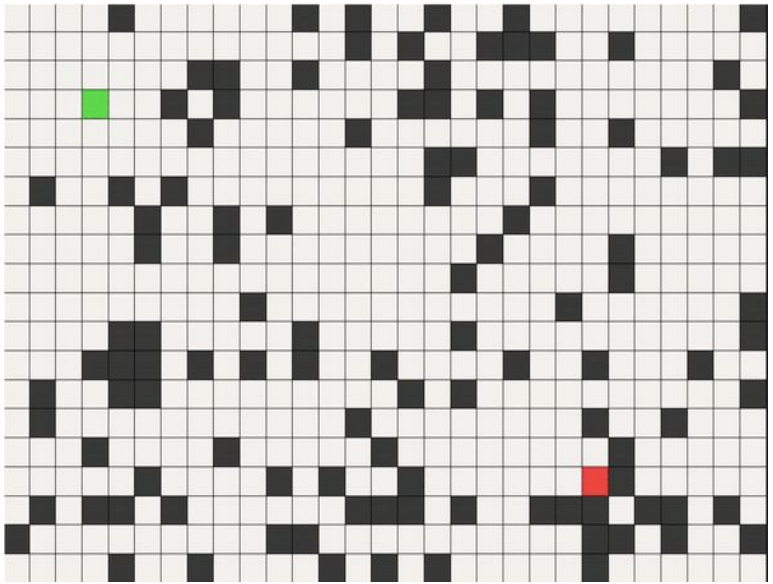




BFS vs DFS
Encontrar una ruta



BFS



DFS

El auto de Cristian Ruz

El auto de **Cristian Ruz** está malo, y se calienta si anda en calles de 5 km o más. Por lo tanto, solo puede transitar por calles cortas (**< 5 km**), descansa un rato y puede seguir transitando por otra calle corta.

Escribe un método que desde un **lugar inicial** entregue todos los **lugares alcanzables** con el auto de **Cristian Ruz**.

Representación en nodos con pesos

```
class Lugar:

  def __init__(self, nombre):
    self.nombre = nombre
    self.calles = list()

  def agregar_calle(self, destino):
    self.calles.append(destino)
```

```
class Ciudad:

  def __init__(self):
    self.lugares = lugares
```

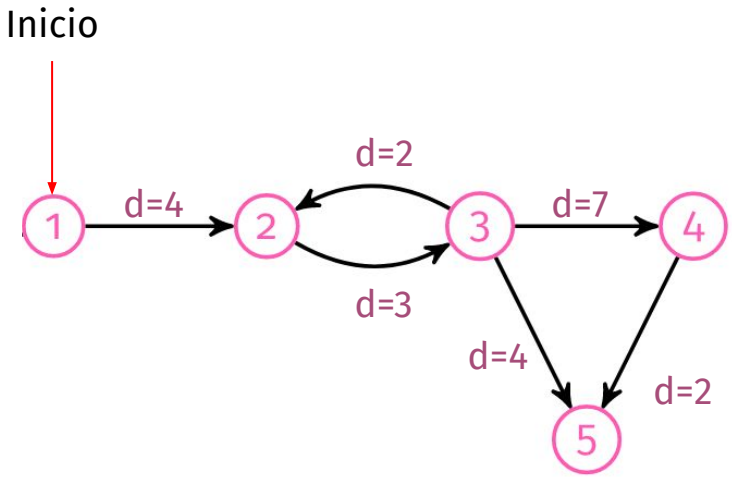
Representación en nodos con pesos

```
class Lugar:

  def __init__(self, nombre):
    self.nombre = nombre
    self.calles = list()

  def agregar_calle(self, distancia, destino):
    self.calles.append(
      (distancia, destino)
    )
```

```
class Ciudad:

  def __init__(self):
    self.lugares = lugares
```

Representación basada en nodos con pesos

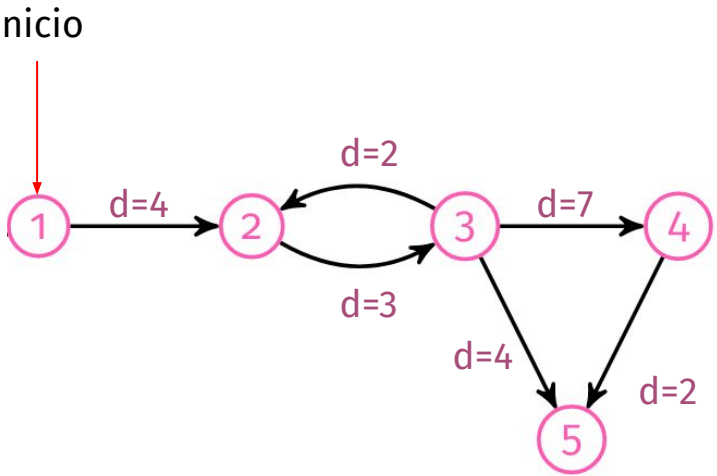
```
class Ciudad:

    def cruz_alcanza(self, inicio):
        pass









        return visitados
```



Representación basada en nodos con pesos

Creamos nuestro *stack* para guardar los nodos a visitar, y un *set* visitados para guardar los lugares ya visitados.

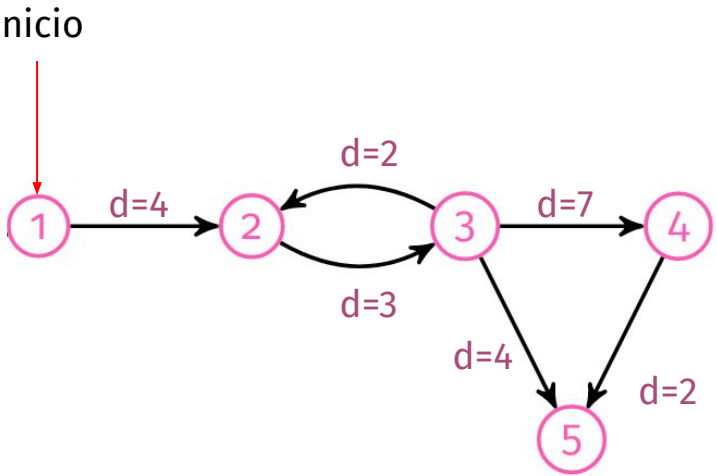
```
class Ciudad:

    def cruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]




        return visitados
```



Representación basada en nodos con pesos

Mientras existan lugares por visitar en el *stack*, sacamos dicho lugar del *stack* para analizarlo.

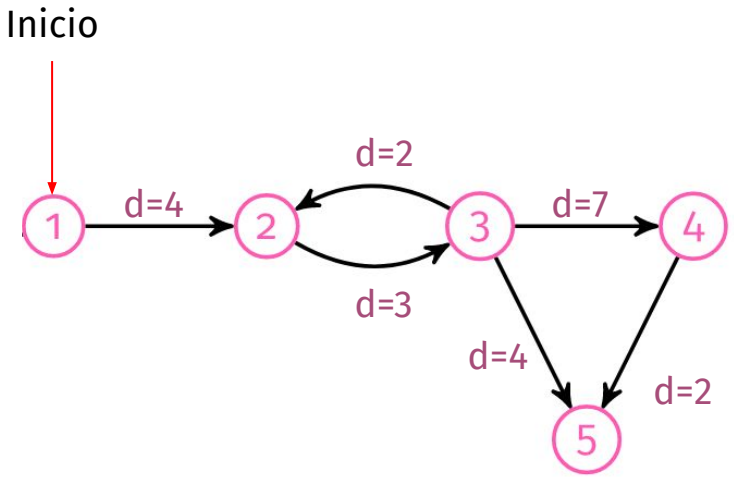
```
class Ciudad:

    def cruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()




        return visitados
```



Representación basada en nodos con pesos

Si el lugar no ha sido visitado, lo agregamos a nuestro *set* de visitados.

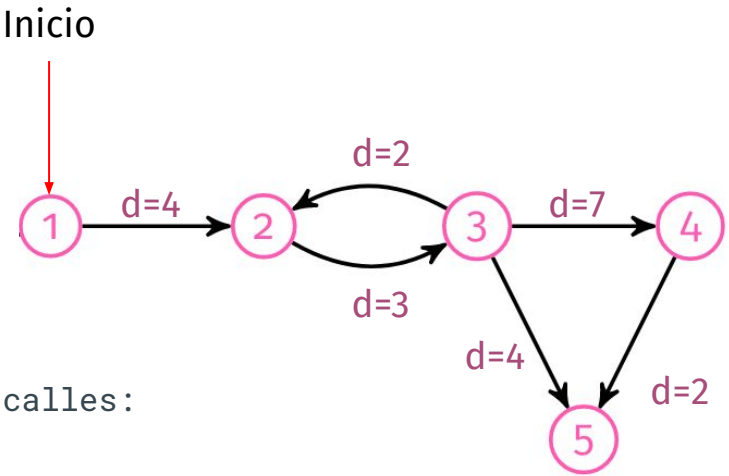
```
class Ciudad:

    def cruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()
            if lugar not in visitados:
                visitados.add(lugar)



        return visitados
```



Representación basada en nodos con pesos

Ahora vamos a analizar los vecinos de ese lugar, si los agregamos a nuestro *stack* de lugares a visitar o no.

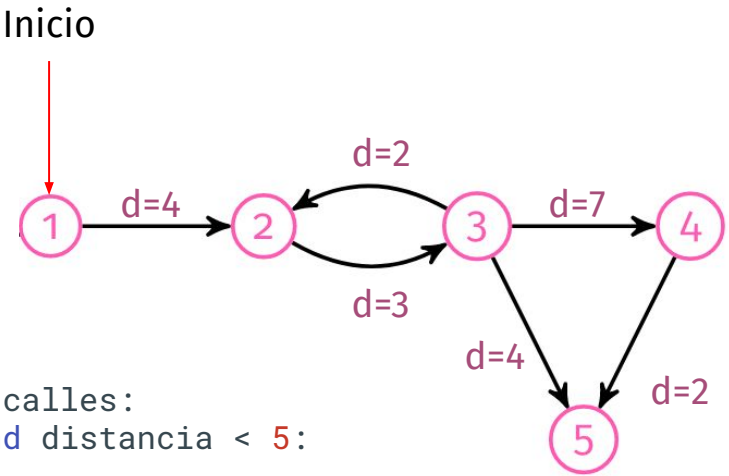
```
class Ciudad:

    def cruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()
            if lugar not in visitados:
                visitados.add(lugar)
                for (distancia, vecino) in lugar.calles:
                    pass

        return visitados
```



Representación basada en nodos con pesos

Si el vecino no ha sido visitado y está a menos de 5 KM, entonces es un lugar visitable y lo agregamos al *stack.*

```
class Ciudad:

    def ruz_alcanza(self, inicio):
        visitados = set()
        stack = [inicio]
        while len(stack) > 0:
            lugar = stack.pop()
            if lugar not in visitados:
                visitados.add(lugar)
                for (distancia, vecino) in lugar.calles:
                    if vecino not in visitados and distancia < 5:
                        stack.append(vecino)
        return visitados
```



Programación Avanzada
IIC2233 2023-2

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Joaquín Tagle - Francisca Cattan

Semana 15 - Estructuras Nodales