

Universidad Autónoma de Nuevo

León

Facultad de Ciencias Físico

Matemáticas

Matemáticas Computacionales

Reporte de Pila – Fila

Alumna: Daniela Carrizales González

Docente: José Anastacio


Reporte

¿Qué es?

Pila:


Se le denomina “clase Pila” a un método que se puede utilizar en Python, entre otros; es un método en el cual ordena datos que posibilita acumular datos y es de tipo LIFO (*Last In First Out*), que quiere decir, “último en entrar, primero en salir”. Además tiene de dos operaciones que son esenciales, las cuales son:


 Apilar, en Python: **push**

 Desapilar, en Python: **pop**

Fila:

Se le denomina “clase Fila” o “Cola” a un método que se puede utilizar en Python, entre otros; se resume que es de tipo FIFO (*First In First Out*), aquí está la diferencia entre “pila” y “cola”, pues FIFO se refiere a que “el primer elemento que entra será también el primero en salir”, se divide los elementos, en inicio y final, ya que, en el inicio se hace la eliminación y en el final una inserción. Asimismo, igual que en “pila” se utiliza:

 Eliminación, en Python: **push**

 Inserción, en Python: **pop**

Grafo:

Por definición tenemos que es una representación simbólica de los componentes de un conjunto, en programación, consiste en conjuntos de vértices y aristas que se está relacionados.

Reporte

Códigos

Pila:

class

Pila:

```
def __init__(self):
    self.items=[]
    def apilar(self, x):
        self.items.append(x)
    def desapilar(self):
        try:
            return self.items.pop()
        except IndexError:
            raise ValueError("La pila está vacía")
    def es_vacia(self):
        return self.items == []
```

Fila:

class

fila:

```
def __init__(self):
    self.fila=[]
def obtener(self):
    return self.fila.pop()
def meter(self,e):
    self.fila.insert(0,e)
    return len(self,fila)
@property
def longitud(self):
    return len(self,fila)
l=fila()
l.meter(1)
l.meter(2)
l.meter(2)
l.meter(3)
l.meter(100)
l.meter(84583)
print(l.longitud)
print(l.obtener())
```

Reporte

Grafo:

```
class Grafo:
```

```
    def __init__(lista):
```

```
        lista.V = set() # un conjunto
```

```
        lista.E = dict() # un mapeo de pesos de aristas
```

```
        lista.vecinos = dict() # un mapeo
```

```
    def agrega(lista, v):
```

```
        lista.V.add(v)
```

```
        if not v in lista.vecinos: # vecindad de v
```

```
            lista.vecinos[v] = set() # inicialmente no tiene nada
```

```
    def conecta(lista, v, u, valor=1):
```

```
        lista.agrega(v)
```

```
        lista.agrega(u)
```

```
        lista.E[(v, u)] = lista.E[(u, v)] = valor # en ambos sentidos
```

```
        lista.vecinos[v].add(u)
```

```
        lista.vecinos[u].add(v)
```

```
    def complemento(lista):
```

```
        comp= Grafo()
```

```
        for v in lista.V:
```

```
            for w in lista.V:
```

```
                if v != w and (v, w) not in self.E:
```

```
                    comp.conecta(v, w, 1)
```

Reporte

return comp

DFS y BFS

Códigos

DFS: Búsqueda en profundidad, pasa por todos los nodos de un grafo de modo estructurado pero no parejo.

```
class
Pila:
    def __init__(self):
        self.items=[]
    def apilar(self, x):
        self.items.append(x)
    def desapilar(self):
        try:
            return self.items.pop()
        except IndexError:
            raise ValueError("La pila está vacía")
    def es_vacia(self):
        return self.items == []
```

class Grafo:

```
    def __init__(lista):

        lista.V = set() # un conjunto

        lista.E = dict() # un mapeo de pesos de aristas

        lista.vecinos = dict() # un mapeo

    def agrega(lista, v):

        lista.V.add(v)

        if not v in lista.vecinos: # vecindad de v

            lista.vecinos[v] = set() # inicialmente no tiene nada

    def conecta(lista, v, u, valor=1):
```

Reporte

```
lista.agrega(v)
```

```
lista.agrega(u)
```

```
lista.E[(v, u)] = lista.E[(u, v)] = valor # en ambos sentidos
```

```
lista.vecinos[v].add(u)
```

```
lista.vecinos[u].add(v)
```

```
def complemento(lista):
```

```
    comp= Grafo()
```

```
    for v in lista.V:
```

```
        for w in lista.V:
```

```
            if v != w and (v, w) not in self.E:
```

```
                comp.conecta(v, w, 1)
```

```
    return comp
```

```
def DFS ( g , ni ):
```

```
    visitados = []
```

```
    f = Pila ()
```

```
    f.meter (ni)
```

```
    while (f.long > 0 ):
```

```
        na = f.obtener ()
```

```
        visitados.append (na)
```

```
        ln = g.vecinos [na]
```

```
        for nodo in ln:
```

```
            if nodo not in visitados:
```

```
                f.meter (nodo)
```

```
    return visitados
```

Reporte

```
desm. = Grafo ()
```

```
desm.conecta (' a ', ' b ')
```

```
desm.conecta (' c ', ' a ')
```

```
desm.conecta (' d ', ' b ')
```

```
DFS (desm, ' a ')
```

```
DFS (desm, ' b ')
```

```
desm.E
```

```
desm v
```

Reporte

BFS: Búsqueda en anchura se usa para pasar por todos los nodos o buscar ciertos elementos, se comienza en la raíz y se analiza todos los vecinos del nodo, después para pasar con los cercanos a otros vecinos.

```
class
fila:
    def __init__(self):
        self.fila=[]
    def obtener(self):
        return self.fila.pop()
    def meter(self,e):
        self.fila.insert(0,e)
        return len(self,fila)
    @property
    def longitud(self):
        return len(self,fila)
l=fila()
l.meter(1)
l.meter(2)
l.meter(2)
l.meter(3)
l.meter(100)
l.meter(84583)
print(l.longitud)
print(l.obtener())
class Grafo:
    def __init__(lista):
        lista.V = set() # un conjunto
        lista.E = dict() # un mapeo de pesos de aristas
        lista.vecinos = dict() # un mapeo
    def agrega(lista, v):
        lista.V.add(v)
        if not v in lista.vecinos: # vecindad de v
```


Reporte

```
lista.vecinos[v] = set() # inicialmente no tiene nada
```

```
def conecta(lista, v, u, valor=1):
```

```
    lista.agrega(v)
```

```
    lista.agrega(u)
```

```
    lista.E[(v, u)] = lista.E[(u, v)] = valor # en ambos sentidos
```

```
    lista.vecinos[v].add(u)
```

```
    lista.vecinos[u].add(v)
```

```
def complemento(lista):
```

```
    comp= Grafo()
```

```
    for v in lista.V:
```

```
        for w in lista.V:
```

```
            if v != w and (v, w) not in self.E:
```

```
                comp.conecta(v, w, 1)
```

```
    return comp
```

```
def BFS (g , ni ):
```

```
    visitados = []
```

```
    f = Fila ()
```

```
    f.meter (ni)
```

```
    mientras que (f.longi > 0 ):
```

```
        na = f.obtener ()
```

```
        visitados.append (na)
```

```
        ln = g.vecinos [na]
```

```
        for nodo in ln:
```

Reporte

```
    if nodo not in visitados:
```

```
        f.meter (nodo)
```

```
    return visitados
```

```
desm = Grafo ()
```

```
desm.conecta ( ' a ' , ' b ' )
```

```
pdesm.conecta ( ' c ' , ' a ' )
```

```
desm.conecta ( ' d ' , ' b ' )
```

```
BFS (desm, ' a ' )
```

```
BFS (desm, ' b ' )
```

```
desm.E
```

```
desm v
```