# Estructuras de datos.

Reporte de la clase de Matemáticas Computacionales.

Pila. Fila. Grafo. BFS. DFS.

*Abstract.*

*El objetivo de este reporte es dar a conocer las distintas estructuras de datos vistas en clase, su relación entre sí, para que sirven y cuál fue la implementación que les dimos, entre algunos otros conceptos. Además de incluir los códigos y algún ejemplo de lo aprendido.*

## Introducción. –

¿Qué es una estructura de datos? Una estructura de datos es una forma particular de organizar datos para que puedan ser utilizados de una manera eficiente, es decir; son un medio para manejar grandes cantidades de datos de manera eficaz.

Existen diferentes estructuras que son adecuadas para una variedad de aplicaciones y/o tareas específicas.

### Pila. –

Una pila es un conjunto de elementos dispuestos unos sobre otros a modo de pilar o columna. Lo que implica que al momento de querer adquirir una “cosa” de ese conjunto se obtiene el último elemento dispuesto.

**Implementación en Python (Código):**

```
#Pila
class pila(object): #quitas el más nuevo
    def __init__(self):
        self.a=[]

    def obtener(self):
        return self.a.pop()

    def meter(self,e):
        self.a.append(e)
    @property
    def longitud(self):
        return len(self.a)

    def __str__(self):
        return str(self.a)

```

Definimos nuestra pila como “class”. Y agregamos las funciones que necesitará para ser llamada una pila como tal.

La función meter en donde utilizamos “append”, nos permite añadir un nuevo elemento a nuestra pila.

La función obtener en donde utilizamos “pop”, extrae de nuestra pila el último elemento. Es decir, el último elemento añadido con la función meter.

Por último, podemos obtener nuestra fila/arreglo con sus elementos actuales.

**Ejemplo:**

```
In [121]: p=pila()

In [122]: p.meter(8)

In [123]: p.meter(2)

In [124]: p.meter("a")

In [125]: p.meter(3)

In [126]: p.meter(2)

In [127]: p.meter("b")

In [128]: print(p)
[8, 2, 'a', 3, 2, 'b']
```

Como puede observarse solo hicimos uso de la función meter.

Ahora, utilizamos la función obtener y veamos que pasa al imprimir nuestra pila.

```
In [127]: p.meter("b")

In [128]: print(p)
[8, 2, 'a', 3, 2, 'b']

In [129]: p.obtener()
Out[129]: 'b'

In [130]: print(p)
[8, 2, 'a', 3, 2]

In [131]: p.obtener()
Out[131]: 2

In [132]: print(p)
[8, 2, 'a', 3]

In [133]: p.obtener()
Out[133]: 3

In [134]: print(p)
[8, 2, 'a']

In [135]:
```

Obtenemos precisamente el último elemento que añadimos y al imprimir nuestra "nueva" pila ya no aparece este elemento.

## Fila. –

Una fila es muy parecida a una pila, con la diferencia que el elemento que "tomaremos" será el primero que añadimos. Como una fila del supermercado, al que atienden primero es al que tiene más tiempo esperando.

**Implementación en Python (Código):**

```
26 #Fila
27 class fila(pila):
28     def obtener(self):
29         return self.a.pop(0)
30
```

Como mencionamos fila es muy parecida a pila es por eso que utilizamos "fila(pila)", es como si las funciones que declaramos anteriormente para pila se le heredaran a fila, pero hay que cambiar una de ellas. Por eso volvemos a definir obtener utilizando "pop (0)" para que obtener el primer elemento añadido.

**Ejemplo:**

```
In [136]: f=fila()

In [137]: f.meter("a")

In [138]: f.meter(1)

In [139]: f.meter("b")

In [140]: f.meter(2)

In [141]: f.meter("c")

In [142]: f.meter(3)

In [143]: print(f)
['a', 1, 'b', 2, 'c', 3]

In [144]: f.obtener()
Out[144]: 'a'

In [145]: print(f)
[1, 'b', 2, 'c', 3]

In [146]: f.obtener()
Out[146]: 1

In [147]: print(f)
['b', 2, 'c', 3]

In [148]:
```

Procedimos igual que con pila. Inicializamos nuestra fila y añadimos elementos utilizando la función meter, pero ahora al usar la función obtener, el elemento obtenido es ‘a’ que fue el primero en introducir.

## Grafo. –

Un grafo es un conjunto de objetos llamados vértices o nodos unidos por aristas, que permiten representar relaciones binarias entre elementos de un conjunto.

**Implementación en Python (Código):**

Entonces, asignaremos vértices y aristas para formar un grafo.

Los vértices son solo elementos y las aristas son pares ordenados de estos elementos.

```
33 #Grafo
34 class grafo:
35     def __init__(self):
36         self.vertices=set()
37         self.E=dict() #para las aristas
38         self.vecinos=dict()
39 
40     def A(self, v): #Agregar
41         self.vertices.add(v)
42         if not v in self.vecinos: #Vecindá de v
43             self.vecinos[v]=set()
44 
45     def C(self, v, u, peso=1): #Concectar
46         self.A(v)
47         self.A(u)
48         self.E[(v,u)]=self.E[(v,u)]=peso
49         self.vecinos[v].add(u)
50         self.vecinos[u].add(v)
51 
52     def __str__(self):
53         return "Aristas= " + str(self.E)+"\nVertices = " +str(self.vertices)
54 
```

Tenemos dos funciones, para los vértices (Recordando que un vecino es aquel que está conectado por una arista con ese elemento) y otra para las aristas. Ahora en esta última, al momento de “conectar elementos” ya estamos agregándolos, entonces utiliza la función agregar.

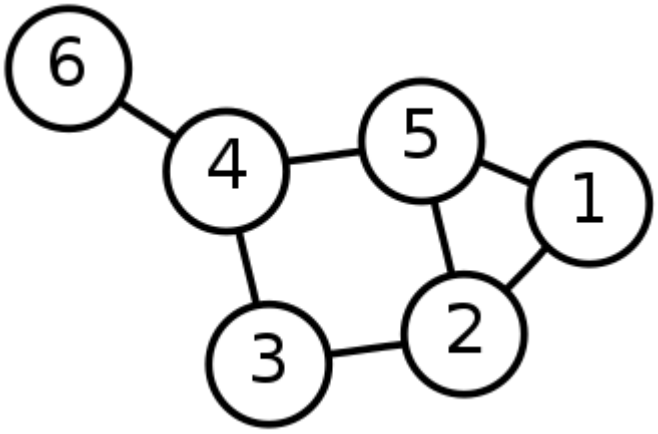
Con la función “str” obtendremos cuantas aristas (junto con su peso) y vértices tiene nuestro grafo.

*Consideramos el peso de todas las aristas igual a 1*

**Ejemplo:**

```
In [153]: g=grafo()

In [154]: g.C(1,2)

In [155]: g.C(2,3)

In [156]: g.C(3,4)

In [157]: g.C(4,5)

In [158]: g.C(4,6)

In [159]: g.C(5,1)

In [160]: g.C(5,2)

In [161]: print(g)
Aristas= {(1, 2): 1, (2, 3): 1, (3, 4): 1, (4, 5): 1, (4, 6): 1, (5, 1): 1, (5, 2): 1}
Vertices = {1, 2, 3, 4, 5, 6}

In [162]:
```

Gráficamente:




A continuación, veremos cómo relacionar las Pilas y Filas con los Grafos.

## BFS. –

Ahora que conocemos el concepto de grafo, suele tener cierta importancia “recorrerlo”, por eso se crean algoritmos que se encarguen de esta tarea específica.

Uno de ellos es “Búsqueda por Anchura” *(en inglés BFS – Breadth First Search),* que se usado frecuentemente sobre árboles, eligiendo algún nodo como raíz y se exploran los vecinos de este nodo.

**Implementación en Python (Código):**

Para esta implementación necesitaremos lo anteriormente visto, un grafo (si no pues que voy a revisar) y una fila.

Aquí es cuando esas estructuras de datos contribuyen a otras.

```
58 #BFS
59 def BFS_n(g, ni):      #ni=vertice
60     visitados=[]       #arreglillo con nodos visitados inicalmente vacio
61     Vvisitar=fila()    #fila con los nodos por visitar
62     Vvisitar.meter( ni )
63     while Vvisitar.longitud > 0:
64         nodo = Vvisitar.obtener()
65         if nodo not in visitados:
66             visitados.append(nodo)
67             for vecino in g.vecinos[nodo]:
68                 Vvisitar.meter(vecino)
69     return visitados
70
```

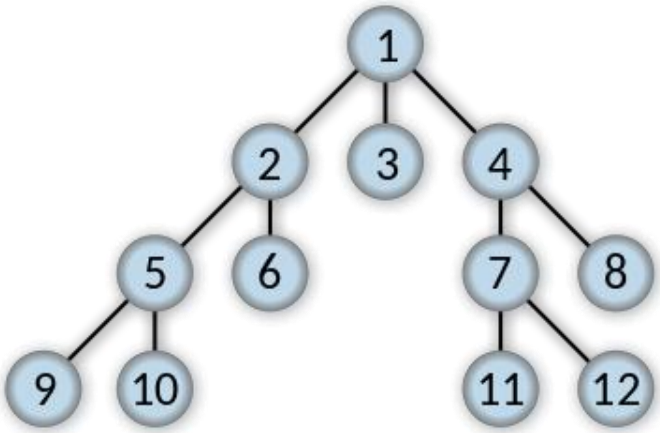
¿Por qué se incorpora el uso de una fila en este código?

Como se mencionó el BFS recorre el grafo según sus vecinos, entonces va empezar con los primeros (las primeras aristas) y una vez acabado con estos vecinos recorrerá a los vecinos de estos. Como si fuera en niveles.

**Ejemplo:**

```
In [165]: g=grafo()

In [166]: g.C(1,2)

In [167]: g.C(1,3)

In [168]: g.C(1,4)

In [169]: g.C(2,5)

In [170]: g.C(2,6)

In [171]: g.C(5,9)

In [172]: g.C(5,10)

In [173]: g.C(4,7)

In [174]: g.C(4,8)

In [175]: g.C(7,11)

In [176]: g.C(7,12)

In [177]: print(g)
Aristas= {(1, 2): 1, (1, 3): 1, (1, 4): 1, (2, 5): 1, (2, 6): 1, (5, 9): 1, (5, 10): 1, (4, 7): 1, (4, 8): 1, (7, 11): 1, (7, 12): 1}
Vertices = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [178]:
```

Este va a ser nuestro grafo, que se ve así:




Como debería de esperarse el recorrido debería ser: 1,2,3,4,5,6,7,8,9,10,11,12.

```
In [177]: print(g)
Aristas= {(1, 2): 1, (1, 3): 1, (1, 4): 1, (2, 5): 1, (2, 6): 1, (5, 9): 1, (5, 10): 1, (4, 7): 1, (4, 8): 1, (7, 11): 1, (7, 12): 1}
Vertices = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [178]: BFS_n(g,1)
Out[178]: [1, 2, 3, 4, 5, 6, 8, 7, 9, 10, 11, 12]

In [179]:
```

Permisos: RW | Fin de línea: CRLF | Codificación: UTF-8 | Línea: 85 | Columna: 5 | Memoria: 54 %

Como puede observarse, le damos de parámetros nuestro grafo y le asignamos ese nodo “raíz”, y evidentemente recorrió el grafo de manera correcta.

## DFS. –

Búsqueda en profundidad *(en inglés DFS o Depth First Search)* al igual que BFS es un algoritmo de búsqueda utilizado para recorrer todos los nodos de un grafo de manera ordenada. Lo que difiere de un algoritmo a otro es que el DFS recorre un vecino y de este recorre a los suyos, así hasta que no quede ninguno, cuando termine vuelve y toma al siguiente vecino de la raíz.

**Implementación en Python (Código):**

Lo que necesitamos será obviamente un grafo que recorrer.

Ya que los algoritmos son muy parecidos, casi podríamos decir que es el mismo código, solo que en vez de usar una fila usaremos una pila.

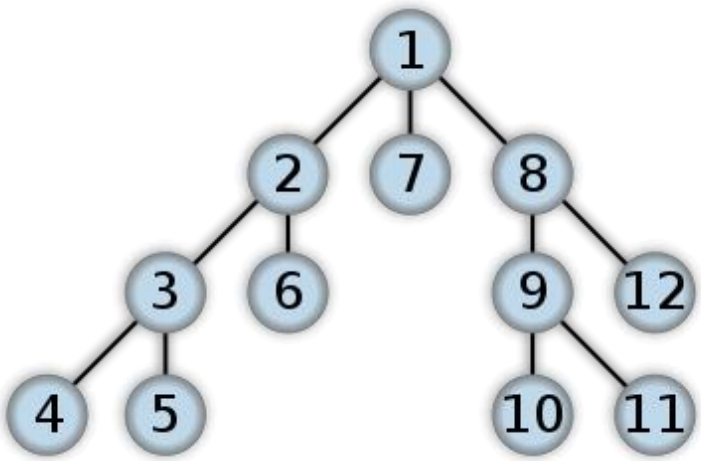
```
71 #DFS
72 def DFS_n(g, ni):
73     visitados=[]      #arreglillo con nodos visitados inicalmente vacio
74     Vvisitar=pila()   #pila con los nodos por visitar
75     Vvisitar.meter( ni )
76     while Vvisitar.longitud > 0:
77         nodo = Vvisitar.obtener()
78         if nodo not in visitados:
79             visitados.append(nodo)
80             for vecino in g.vecinos[nodo]:
81                 Vvisitar.meter(vecino)
82     return visitados
```

**Ejemplo:**

Este va a ser nuestro grafo:

```
In [179]: g=grafo()

In [180]: g.C(1,2)

In [181]: g.C(1,7)

In [182]: g.C(1,8)

In [183]: g.C(2,3)

In [184]: g.C(2,6)

In [185]: g.C(8,9)

In [186]: g.C(8,12)

In [187]: g.C(3,4)

In [188]: g.C(3,5)

In [189]: g.C(9,10)

In [190]: g.C(9,11)

In [191]: print(g)
Aristas= {(1, 2): 1, (1, 7): 1, (1, 8): 1, (2, 3): 1, (2, 6): 1, (8, 9): 1, (8, 12): 1, (3, 4): 1, (3, 5): 1, (9, 10): 1, (9, 11): 1}
Vertices = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [191]:
```

Gráficamente se ve así:



El recorrido esperado sería: 1,2,3,4,5,6,7,8,9,10,11,12.

```
In [191]:

In [192]: print(g)
Aristas= {(1, 2): 1, (1, 7): 1, (1, 8): 1, (2, 3): 1, (2, 6): 1, (8, 9): 1, (8, 12): 1, (3, 4): 1, (3, 5): 1, (9, 10): 1, (9, 11): 1}
Vertices = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

In [193]: DFS_n(g,1)
Out[193]: [1, 7, 2, 6, 3, 5, 4, 8, 9, 11, 10, 12]

In [194]:
```

El recorrido fue es esperado, pero en diferente orden, pero siguió el algoritmo correctamente. Igualmente, el vértice que tomo como raíz fue el 1.

# Conclusiones. –

Lo importante de estas estructuras de datos es su aplicación/implementación ya sea para cierta tarea o para complementar otra estructura y así sea mucho más eficiente; como la pila y fila, que en un principio se podría creer que no tenían ninguna relación con los grafos, hasta que surge la tarea de recorrer estos dichos grafos.

Son estructuras y algoritmos bastante entretenidos, sobre todo la elaboración de un grafo y el cómo recórrelo. Que quiero decir con esto, para empezar, hay dos maneras de hacerlo, por BFS y DFS, pero además tú le asignas que vértice tomar como “raíz” es decir, podría recorrerlo de maneras diferentes con un solo algoritmo.

Eso fue lo que más me gusto de los ejercicios realizados, lo didácticos que son.

Ya no se me olvidó subirlo en PDF.