# Reporte de estructuras de datos.

María Luisa Borrego Riojas

Universidad Autónoma de Nuevo León

Facultad de Ciencias Físico Matemáticas

Matemáticas Computacionales

En este reporte se buscará definir y ejemplificar las estructuras de pila, fila y los grafos. Cada una de ellas se llevará a cabo para los algoritmos de Depth First Search (DPS) y Breadth First Search (BFS).

1. **Pila.** Una pila es una especie de contenedor de objetos que se insertan y eliminan siguiendo el orden desde la última entrada hasta la primera salida. Es importante señalar que sólo se permiten dos operaciones: Meter un elemento a la pila y extraerlo de la pila. Esta estructura de datos es de acceso limitado, pues los elementos que se agregan o extraen sólo puede ser de la parte superior. Un ejemplo de esto en la vida real es como un montón de platos apilados; sólo se puede agregar el plato encima de los demás, así como sólo se puede retirar el plato de arriba.

   Ahora, el pseudocódigo para esta estructura de datos es:

```
class pila (object):
      def __init__(self):
            self.x=[]
      def obtener (self):
            return self.x.pop()
      def meter(self,y):
            self.x.append(y)
            return len(self.x)
      @property
            def long(self):
            Return len(self.a)
```

2. **Fila.** Una fila es un contenedor de objetos acomodados de forma lineal que se insertan y eliminan según el principio first-in first-out (FIFO), esto es, el primero en entrar es el primero en salir. Una ejemplificación de esto es una línea de alumnos en la cafetería de la facultad, en donde se atiende al que lleva más tiempo esperando.

   A continuación, el pseudocódigo para Python:

```
class fila(objecto):
      def __init__(self):
            self.x=[]
      def obtener (self):
            return self.x.pop(0)
      def meter(self,y):
            self.x.append(y)
            return len(self.x)
      @property
      def longitud(self):
            return len(self.x)
```

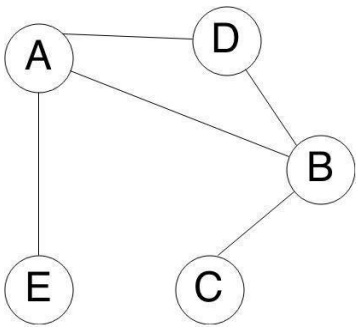
La diferencia entre las pilas y filas está en la eliminación. En una pila se elimina el elemento que fue agregado más recientemente, mientras que en una fila se elimina el menos reciente.

3. **Grafo.** Se le llama así a un conjunto de nodos (o vértices, si así se prefiere) que se relacionan a través de arcos (o aristas). Los grafos normalmente nos sirven para representar relaciones entre cosas o personas, así como rutas de cualquier tipo. Además de esto, claramente tiene otras aplicaciones.

Ahora, el pseudocódigo:

```
Class grafo(object):
      def__init__(self):
            self.nodo=set()
            self.arco=dict()
            self.vecinos=dict()
      def add(self,x):
            self.nodo.add(x)
            if not x in self.vecinos:
                  self.vecinos[x]=set()
      def connect(self,y,x,peso=1):
            self.add(y)
            self.add(x)
            self.nodo[(y,x)]=self.nodo[(x,y)]=peso
            self.vecinos[y].add(x)
            self.vecinos[x].add(y)
      @property
      def complement(self):
            c=grafo()
            for a in self.nodo:
                  for b in self.nodo:
                        if a!=b and (a,b) not in self.arco:
                              c.connect(a,b,1)
```

4. **Depth First Search (DFS).** Es un algoritmo que se usa para recorrer un grafo.




Primero se selecciona un nodo de este y se explora lo más lejos posible en una rama para regresar a un punto fijo. DFS generalmente se usa para cuestiones de conectividad. Tiene una complejidad de **O(N+E)** donde N es el número total de nodos y E es el número total de aristas.

Para representar el grafo de la imagen, el código en Python es:

```
graph={ ‘A’ :[ ‘E’ ,’ B’ ,’ D’ ],
        ‘B’ :[ ‘A’ ,’ D’ ,’ C’ ],
        ‘C’ :[ ‘B’ ],
        ‘D’ :[ ‘A’ ,’ B’ ],
        ‘E’ :{ ‘A’ ]}
```

Una vez que tenemos la lista, aplicamos DFS. Primero seleccionamos un nodo. Hay que recordar que se recorren los nodos hacia abajo.

Los pasos por realizar son:

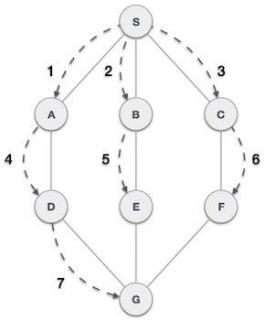
1) Seleccionar un nodo. Agregarlo a la lista de visitados.
2) Buscar nodos adyacentes. Añadir a la lista aquellos que no han sido visitados
3) “pop” el primer nodo y repetir los primeros dos pasos.

Esto es, en pseudocódigo:

```
graph={ 'A':['E','B','D'],
        'B':['A','D','C'],
        'C':['B'],
        'D':['A','B'],
        'E':['A']}

def dfs(graph,s):
  stack=[]
  visited=[]
  stack=[s]
  while stack:
      node=stack.pop()
      if node not in visited:
            visited.append(node)
            stack=stack+graph[node]
  return visited
```

5. **Breadth First Search (BFS)**. Este algoritmo de búsqueda recorre el grafo en un movimiento horizontal y utiliza una pila para recordar cuál es el siguiente vértice para iniciar una búsqueda, en caso de que una de las iteraciones llegue a su fin.




Tomemos como ejemplo la imagen de la izquierda. En este caso el algoritmo BFS atraviesa el grafo en el orden A,B,E,F,C,G,D.

Las reglas para este algoritmo son:

1) Se visita el vértice adyacente que no se ha visitado. Después de esto ya se puede marcar como visitado e insertar en la pila.
2) En caso de no encontrarse ningún vértice adyacente, se quita el primer vértice de la pila.
3) Se repite la regla 1) y 2) hasta que la pila esté vacía.

El pseudocódigo es:

```
def bfs(grafo,i):
        visited=[i]
        bsq=fila()
        bsq.meter(i)
        while bsq.long>0:
            act=bsq.obtener()
            vecinos=grafo.vecinos[act]
                  for x in vecinos:
                        if x not in visited:
                              visited.append(w)
                              bsq.meter(x)
        return visited
```