Reporte de estructuras de datos

Haydee Judith Arriaga Ponce

Matricula: 1659539

l.judithmat@gmail.com

https://github.com/haydee10arriaga/1659539_Mat.Com

6 de octubre de 2017

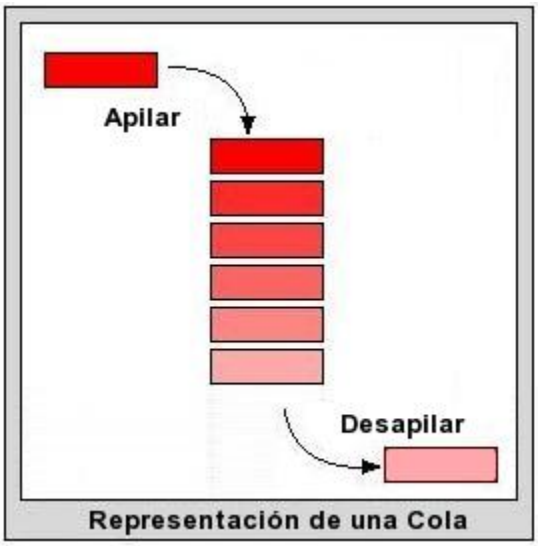
En el siguiente reporte se presenta brevemente el concepto de las principales estructuras de datos Pilas, Filas y Grafos, que se utilizan generalmente para simplificar ciertas operaciones de programación. Estas estructuras pueden implementar algoritmos BFS, DFS.

FILA O COLA

“El primero que entra es el primero que sale”. Un ejemplo de ello son las filas que se hacen en los supermercados, cines, bancos, etc.

Es una estructura de datos, en la cual solo se pueden aplicar dos operaciones: colocar un elemento al final, o quitar un elemento del principio, es decir se entiende por fila una estructura en la que se añaden nuevos ítems en un extremo y se suprimen ítems viejos en el opuesto. En realidad, me pareció muy práctico el hecho de que a partir de un arreglo me devuelve el primer elemento que se ingresó y deja de ser el primero ya que si se llama de nuevo a la función se toma en cuenta el siguiente elemento y así sucesivamente.

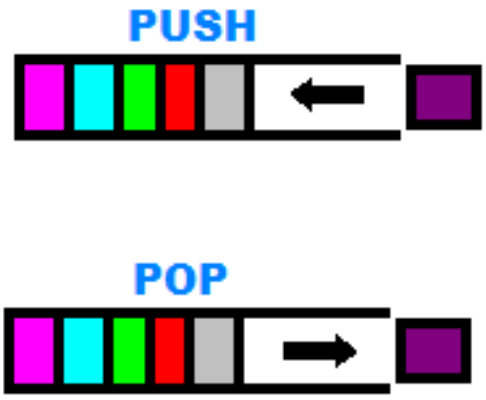
```
 8 #Haydee Judith Arriaga
 9 class Fila(object):
10     def __init__(self):
11         self.fila=[]
12     def obtener(self):
13         return self.fila.pop(0)
14     def meter(self,e):
15         self.fila.append(e)
16         return len(self.fila)
17     @property
18     def longitud(self):
19         return len(self.fila)
20 
21 
22 fila=Fila()
23 fila.meter(5)
24 fila.meter('fila')
25 fila.meter("hello")
26 print(fila.longitud)
27 print(fila.obtener())
28 print(fila.obtener())
29 print(fila.obtener())
30 print(fila.longitud)
```

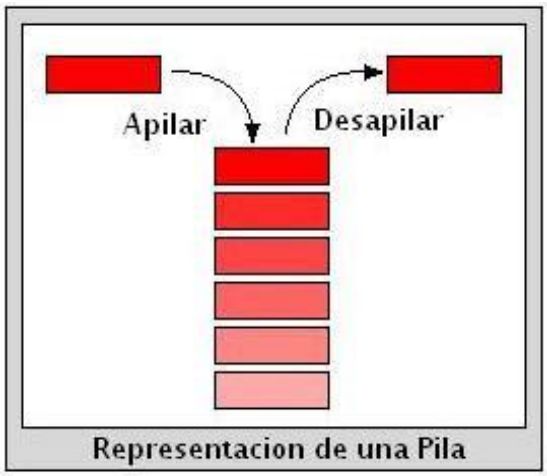


PILA

Las pilas son estructuras de datos que tienes dos operaciones básicas: push (para insertar un elemento) y pop (para extraer un elemento). Su característica fundamental es que al extraer se obtiene siempre el último elemento que acaba de insertarse. Por esta razón también se conocen como estructuras de datos LIFO (del inglés Last In First Out). Una posible implementación mediante listas enlazadas sería insertando y extrayendo siempre por el principio de la lista. Gracias a las pilas es posible el uso de la recursividad.

Las pilas se utilizan en muchas aplicaciones que utilizamos con frecuencia. Por ejemplo, la gestión de ventanas en Windows (cuando cerramos una ventana siempre recuperamos la que teníamos detrás). Otro ejemplo es la evaluación general de cualquier expresión matemática para evitar tener que calcular el número de variables temporales que hacen falta.



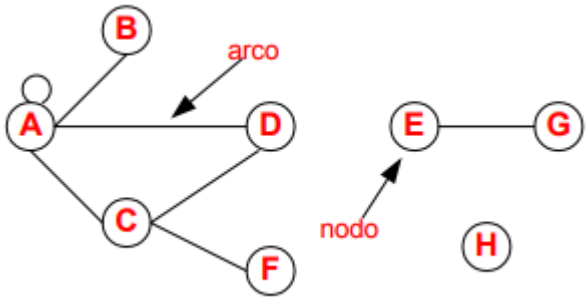


```python
class Pila(object):
    def __init__(self):
        self.pila=[]
    def obtener(self):
        return self.pila.pop()
    def meter(self,e):
        self.pila.append(e)
        return len(self.pila)
    @property
    def longitud(self):
        return len(self.pila)

p=Pila()
p.meter(5)
p.meter("pila")
p.meter("hello")
print(p.longitud)
print(p.obtener())
print(p.obtener())
print(p.obtener())
print(p.longitud)
```

GRAFO

Un grafo está formado por un conjunto de nodos (o vértices) y un conjunto de arcos. Cada arco en un grafo se especifica por un par de nodos. El conjunto de nodos es {A, B, C, D, F, G, H} y el conjunto de arcos {(A, B), (A, D), (A, C), (C, D), (C, F), (E, G), (A, A)} para el siguiente grafo:



TERMINOLOGÍA

*.-Al número de nodos del grafo se le llama orden del grafo.

*.-Un grafo nulo es un grafo de orden 0 (cero).

*.-Dos nodos son adyacentes si hay un arco que los une.

*.-En un grafo dirigido, si A es adyacente de B, no necesariamente B es adyacente de A

*.-Camino es una secuencia de uno o más arcos que conectan dos nodos.

*.-Un grafo se denomina conectado cuando existe siempre un camino que une dos nodos cualesquiera y desconectado en caso contrario.

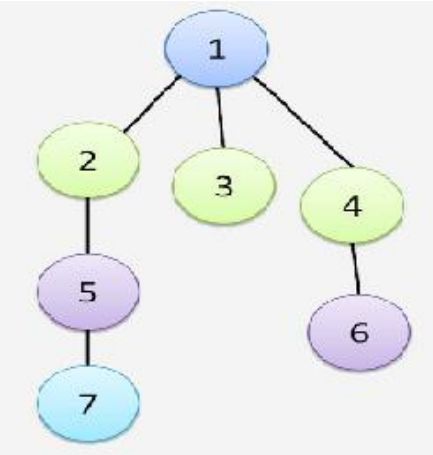
*.-Un grafo es completo cuando cada nodo está conectado con todos y cada uno de los nodos restantes.

*.-El camino de un nodo así mismo se llama ciclo.

```
 7
 8 class Grafo(object):
 9     def __init__(self):
10         self.Vertices=set()
11         self.Aristas=dict()
12         self.Vecinos=dict()
13
14     def Agrega(self,V):
15         self.Vertices.add(V)
16         if not V in self.Vecinos:
17             self.Vecinos[V]=set()
18
19     def Conecta(self,U,V,peso=1):
20         self.Agrega(U)
21         self.Agrega(V)
22         self.Aristas[(U,V)]=self.Aristas[(V,U)]=peso
23         self.Vecinos[U].add(V)
24         self.Vecinos[V].add(U)
25
26     @property
27     def Complemento(self):
28         comp=Grafo()
29         for A in self.Vertices:
30             for B in self.Vertices:
31                 if A!=B and (A,B) not in self.Aristas:
32                     comp.Conecta(A,B,1)
33 g=Grafo()
34 g.Conecta('a','b')
35 g.Conecta('a','c')
36 g.Conecta('d','e')
37 g.Conecta('f','g')
38 g.Conecta('a','e')
```

BFS Algoritmo de Búsqueda en Anchura

La búsqueda en anchura supone que el recorrido se haga por niveles. Para entender más fácilmente de que se trata, hemos indicado en la siguiente imagen un grafo ejemplo en donde cada color representa un nivel, tomando como raíz o nodo inicial el que tiene el número 1. El recorrido se hará en orden numérico de forma consecutiva hasta llegar al nodo número 7.




```
45
46 def BFS(graph,ini):
47     visitados=[ini]
48     F=Fila()
49     F.meter(ini)
50     while (F.longitud>0):
51         act=F.obtener()
52         Vecinos=graph.Vecinos[act]
53         for nodo in Vecinos:
54             if nodo not in visitados:
55                 visitados.append(nodo)
56                 F.meter(nodo)
57     return visitados
```

La estrategia que usaremos para garantizar este recorrido es utilizar una cola que nos permita almacenar temporalmente todos los nodos de un nivel, para ser procesados antes de pasar al siguiente nivel hasta que la cola esté vacía.

Cada vez que visitamos un nodo, lo desencolamos e imprimimos por pantalla el valor del nodo para ir indicando el recorrido. Luego agregamos a la cola todos los nodos del siguiente nivel y los marcamos como visitados antes de comenzar el ciclo de nuevo, en el que procesaremos estos nuevos nodos que hemos agregado a la cola.

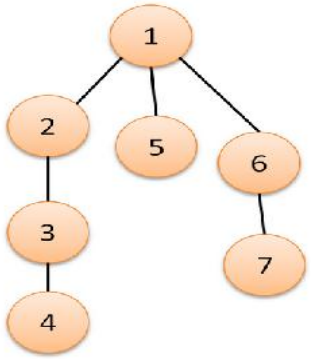
Si encontramos el elemento buscado, la función BFS retornará al “main” e imprimirá entre comillas simples el valor del elemento buscado.

Podríamos hacer otro tipo de mensaje para indicar que el elemento ha sido encontrado, calcular el número de nodos que fueron visitados o incluso generar un mensaje especial en el caso de que el elemento no haya sido encontrado. Por el momento la función ha quedado lo más sencilla posible para enfocarnos únicamente en cómo hacer el recorrido.

```
45
46 def BFS(graph,ini):
47     visitados=[ini]
48     F=Fila()
49     F.meter(ini)
50     while (F.longitud>0):
51         act=F.obtener()
52         Vecinos=graph.Vecinos[act]
53         for nodo in Vecinos:
54             if nodo not in visitados:
55                 visitados.append(nodo)
56                 F.meter(nodo)
57     return visitados
```

DFS Algoritmo de Búsqueda en Profundidad

En el caso de la búsqueda en profundidad lo que se quiere es recorrer desde la raíz hasta los nodos extremos u hojas por cada una de las ramas. En este caso los niveles de cada nodo no son importantes. En la siguiente imagen podemos ver el orden en que se hace el recorrido desde el nodo raíz, indicado con el número uno, hasta el nodo número siete.



La forma más intuitiva de hacer este algoritmo es de forma recursiva, de lo contrario tendríamos que usar en lugar de una cola una pila, pero con la recursión nos ahorramos la necesidad de utilizar esta estructura explícitamente y en lugar de ello nos valemos de la pila de recursión.

El orden en que se eligen las ramas en un recorrido DFS está determinado por el tipo de recorrido de procesamiento de árbol que se haya elegido, estos pueden ser:

Pre-orden: Se procesa primero la raíz, luego la rama izquierda y luego las ramas siguientes hasta llegar a la que se encuentra más a la derecha.

Post-orden: Se procesa el árbol desde las ramas izquierdas hasta la que se encuentra más a la derecha. Finalmente se procesa el nodo raíz

Simétrico o In-orden: Se procesa la rama de la izquierda, luego el nodo raíz y luego la rama derecha.

```
45
46 def DFS(graph,ini):
47     visitados=[]
48     F=Pila()
49     F.meter(ini)
50     while (F.longitud>0):
51         act=F.obtener()
52         if act in visitados:
53             continue
54         visitados.append(act)
55         Vecinos=graph.Vecinos[act]
56         for nodo in Vecinos:
57             if nodo not in visitados:
58                 F.meter(nodo)
59     return visitados
```

BFS Y DFS

```
#Haydee Judith Arriaga
class Fila(object):
    def __init__(self):
        self.fila=[]
    def obtener(self):
        return self.fila.pop(0)
    def meter(self,e):
        self.fila.append(e)
        return len(self.fila)
    @property
    def longitud(self):
        return len(self.fila)


class Grafo(object):
    def __init__(self):
        self.Vertices=set()
        self.Aristas=dict()
        self.Vecinos=dict()

    def Agrega(self,V):
        self.Vertices.add(V)
        if not V in self.Vecinos:
            self.Vecinos[V]=set()

    def Conecta(self,U,V,peso=1):
        self.Agrega(U)
        self.Agrega(V)
        self.Aristas[(U,V)]=self.aristas[(V,U)]=peso
        self.Vecinos[U].add(V)
        self.Vecinos[V].add(U)

    @property
    def Complemento(self):
        comp=Grafo()
        for A in self.Vertices:
            for B in self.Vertices:
                if A!=B and (A,B) not in self.Aristas:
                    comp.Conecta(A,B,1)

def BFS(graph,ini):
    visitados=[ini]
    F=Fila()
    F.meter(ini)
    while (F.longitud>0):
        act=F.obtener()
        Vecinos=graph.Vecinos[act]
        for nodo in Vecinos:
            if nodo not in visitados:
                visitados.append(nodo)
                F.meter(nodo)
    return visitados

graph=Grafo()
graph.Conecta(1,2)
graph.Conecta(2,3)
graph.Conecta(3,4)
graph.Conecta(4,5)
graph.Conecta(1,6)
graph.Conecta(6,7)
graph.Conecta(7,8)
graph.Conecta(8,9)
print(BFS(graph,1))
#[1, 2, 6, 3, 7, 4, 8, 5, 9]
```

```
#Haydee Judith Arriaga
class Pila(object):
    def __init__(self):
        self.pila=[]
    def obtener(self):
        return self.pila.pop()
    def meter(self,e):
        self.pila.append(e)
        return len(self.pila)
    @property
    def longitud(self):
        return len(self.pila)


class Grafo(object):
    def __init__(self):
        self.Vertices=set()
        self.Aristas=dict()
        self.Vecinos=dict()

    def Agrega(self,V):
        self.Vertices.add(V)
        if not V in self.Vecinos:
            self.Vecinos[V]=set()

    def Conecta(self,U,V,peso=1):
        self.Agrega(U)
        self.Agrega(V)
        self.Aristas[(U,V)]=self.aristas[(V,U)]=peso
        self.Vecinos[U].add(V)
        self.Vecinos[V].add(U)

    @property
    def Complemento(self):
        comp=Grafo()
        for A in self.Vertices:
            for B in self.Vertices:
                if A!=B and (A,B) not in self.Aristas:
                    comp.Conecta(A,B,1)

def DFS(graph,ini):
    visitados=[]
    F=Pila()
    F.meter(ini)
    while (F.longitud>0):
        act=F.obtener()
        if act in visitados:
            continue
        visitados.append(act)
        Vecinos=graph.Vecinos[act]
        for nodo in Vecinos:
            if nodo not in visitados:
                F.meter(nodo)
    return visitados
graph=Grafo()
graph.Conecta(1,2)
graph.Conecta(2,3)
graph.Conecta(3,4)
graph.Conecta(4,5)
graph.Conecta(1,6)
graph.Conecta(6,7)
graph.Conecta(7,8)
graph.Conecta(8,9)
print(DFS(graph,1))
#[1, 6, 7, 8, 9, 2, 3, 4, 5]
```

CONCLUSION:

Principalmente el maestro gracias al apoyo que brindo comprendimos y llevamos a la práctica como trabajo las estructuras de datos filas, pilas y grafos, con las cuales implementamos algoritmos como BFS y DFS al concluir pude comprender la deferencia entre los dos algoritmos.

Al trabajar con fila y pila son casi idénticas difieren al cambiar el método de obtener y en el nombre se podría decir que hacen lo contrario una de la otra. Fila es “el primero que entra es el primero que sale” y por el contrario Pila “se obtiene siempre el último elemento que acaba de insertarse”.

En cambio, realizar el grafo es más fácil y practico si ya cuento con fila y pila porque se complementaría con lo que se obtuvo en clase de la página que consultamos de la maestra Elisa uanl la cual la proporciono el maestro fue de gran ayuda de ahí me base para llevar a cabo el grafo que para mí consiste principalmente para búsquedas o en este caso aplico para búsqueda ya que vimos los grafos pero lo más simple lo más sencillo de ello la búsqueda consiste ya sea por amplitud o profundidad que ya es cuando se implementan los algoritmos porque ya con las 3 estructuras de datos el maestro siguió con la implementación de los algoritmos BFS y DFS.

Ambas técnicas constituyen métodos sistemáticos para visitar todos los vértices y arcos del grafo, exactamente una vez y en un orden específico predeterminado, por lo cual podríamos decir que estos algoritmos simplemente nos permiten hacer recorridos controlados dentro del grafo con algún propósito.

Siendo la búsqueda una de las operaciones más sencillas y elementales en cualquier estructura de datos, se han estandarizado el uso de estos algoritmos para ello, por lo que se conocen como algoritmos de búsqueda. Sin embargo, es importante resaltar que pueden utilizarse para muchísimas otras operaciones con grafos que no necesariamente incluyan la búsqueda de algún elemento dentro del

grafo. La diferencia entre BFS y DFS es el orden en que ha de recorrerse el grafo, para algunos problemas no tiene ninguna relevancia cuál de los dos algoritmos ha de aplicarse, pero en otros casos esta elección es crucial. Me parece muy interesante todo este trabajo ya que a pesar de empezar desde lo esencial lo simple da pie a obtener cada quien interés por llegar a saber en un futuro cuando tenemos la oportunidad de aplicar esto y desarrollar más hasta lo más complejo ya que en la carrera de matemáticas casi no le damos la importancia debida a este tipo de materias por tener que ver con la programación la mayoría le saca la vuelta lo cual no me parece bien me agrada poder seguir adquiriendo más conocimientos relacionados a esto para llegar en un momento dado a realizar proyectos donde se implementa todo esto.