Reporte de practica: Estructuras de datos

Nombre: Diana Carolina Lozoya Pérez **Matricula:**1725012

Materia: Matemáticas Computacionales **Grupo:**001 **Fecha:** 06/10/17

Resumen: Este reporte tiene como objetivo describir el concepto de estructura de datos basándose en los ejemplos de que es y cómo programar pila y fila, de manera similar con definir un grafo y así mismo crear un BFS o DFS para aplicar lo anteriormente programado.

1. Fila

Una fila es una estructura de datos que permite almacenar datos en el orden *FIFO* (first-in ,first-out) es decir, Primero en Entrar, Primero en Salir. Además, la recuperación de los datos es hecha en el orden en que son insertados.

Para definir esta clase, seguiremos los siguientes pasos:

a) *__init__:* inicializa una fila nueva, vacía.

b) *meter:* agrega un nuevo elemento a la fila, el cual será el puesto en la última posición de la estructura.

c) *obtener:* regresa el primer elemento para mostrarlo, y después lo elimina.

d) *longitud*: es una propiedad de fila y se define como tal, esta devuelve la dimensión del arreglo, es decir el número de elementos.

Ejemplo en Python:

```
#definir 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 longuitud(self):
        return len(self.fila)
```

2. Pila

Una pila es una estructura de datos, en la cual las inserciones y eliminaciones se hacen por un extremo, que es la parte superior o análogamente ‘’la cima’’. Esta estructura es conocida como un *LIFO* (last-in, first-out) es decir, ultimo en entrar, primero en salir, lo cual básicamente explica su funcionamiento.

Para definir esta clase, seguiremos los siguientes pasos:

e) *__init__:* inicializa una pila nueva, vacía.

f) *meter:* agrega un nuevo elemento a la pila.

g) *obtener:* elimina el tope de la pila y lo devuelve. El elemento que se devuelve es siempre el último que se agrega.

h) *longitud*: es una propiedad de pila y se define como tal, esta devuelve la dimensión del arreglo, es decir el número de elementos.

Ejemplo en Python:

```
#definir Pila
class Pila:
    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 longuitud(self):
        return len(self.pila)
```

3. Grafo

Un grafo es un conjunto de *n* vértices, los cuales están conectados entre sí mediante *m* aristas, análogamente, lo podemos visualizar como un árbol genealógico, las constelaciones, relaciones de amistades, etc. Ya que los vértices en un grafo se suelen dibujar como círculos y las aristas como líneas que les conectan uno al otro.

Para definir o construir esta clase, seguiremos los siguientes pasos:

a) *__init__:* inicializa el conjunto de vértices y aristas, así como los ‘’vecinos’’ que son los vértices que están unidos mediante una arista a otro vértice.

b) *conecta:* de los vértices que tenemos declarados, definimos a ‘’peso’’ como un valor que muestra las aristas entre los vértices tomados.

c) *agrega:* aquí se incluyen los vértices al grafo, en caso de no estar, se inicializa a los ‘’vecinos’’ como un arreglo vacío.

d) *complemento*: es una propiedad, que regresa un grafo, con los vértices dados originalmente.

Ejemplo en Python:

```
#class grafo
class Grafo:
    def __init__(self):
        self.V=set()
        self.E=dict()
        self.vecinos=dict()

    def agrega(self,v):
        self.V.add(v)
        if not v in self.vecinos:
            self.vecinos[v]=set()

    def conecta(self,v,u,peso=1):
        self.agrega(v)
        self.agrega(u)
        self.E[(v,u)]= self.E[(u,v)]=peso
        self.vecinos[v].add(u)
        self.vecinos[u].add(v)

    def complemento(self):
        comp=Grafo()
        for v in self.V:
            for w in self.V:
                if v !=w and (v,w) not in self.E:
                    comp.conecta(v,w,1)
        return comp
```

4. BFS

El **Breadth first search (**anchura de búsqueda inicial**)** es uno de los algoritmos más fáciles para buscar un gráfico.

Dado un gráfico y un vértice de partida, una anchura de la primera búsqueda procede explorando los bordes en el gráfico para encontrar todos los vértices en el gafo para el cual hay un camino del vértice. Lo notable de una primera búsqueda de amplitud es que encuentra *todos* los vértices que están a una distancia k del vértice antes de encontrar *cualquier* vértice que sea una distancia k+1k+1.

Una buena manera de visualizar lo que hace el primer algoritmo de búsqueda es imaginar que está construyendo un árbol, un nivel del árbol a la vez. Una primera búsqueda de amplitud añade todos los hijos del vértice inicial antes de que empiece a descubrir a alguno de los nietos.

Ejemplo en Python:

*Haciendo uso de los códigos de fila y grafo anteriores, el código del BFS quedaría de la siguiente manera:

```
#definir bfs
def bfs(grafo,ni):
    visitados=[ni]
    f=Fila()
    f.meter(ni)
    while f.longuitud>0:
        na=f.obtener()
        vecinos= grafo.vecinos[na]
        for nodo in vecinos:
            if nodo not in visitados:
                visitados.append(nodo)
                f.meter(nodo)
    return visitados
```

5. DFS

El **Depth-Fisrt Search (**Primera búsqueda por profundidad**)**, explora posibles vértices (de una raíz suministrada) por cada rama antes de retroceder. Esta propiedad permite que el algoritmo se implemente sucintamente tanto en forma iterativa como recursiva.

La implementación a continuación utiliza la estructura de datos de pila para generar y devolver un conjunto de vértices que son accesibles dentro del componente conectado a los temas. Utilizando la sobrecarga de Python del operador de substracción para eliminar elementos de un conjunto, podemos agregar sólo los vértices adyacentes no visitados.

Ejemplo en Python:

*Haciendo uso de los códigos de pila y grafo anteriores, el código del DFS quedaría de la siguiente manera:

```
#definir dfs
def dfs(grafo,ni):
    visitados=[ni]
    f=Pila()
    f.meter(ni)
    while f.longuitud>0:
        na=f.obtener()
        vecinos= grafo.vecinos[na]
        for nodo in vecinos:
            if nodo not in visitados:
                visitados.append(nodo)
                f.meter(nodo)
    return visitados
```

6. Conclusiones

En lo personal, batalle un poco al principio con las estructuras de fila y pila, entiendo que hacen, pero al momento de programarlas se venía lo ‘’complicado’’ por así decirlo, he llevado programación en C, sin embargo, no había visto estructuras como lo son pila y fila, por lo que es comprensible que me halla costado un poco.
Estas estructuras, las programe siguiendo las instrucciones del profesor de la clase, quien nos explicó paso por paso lo que teníamos que hacer y además menciono cual era la manera correcta y sencilla de programar una pila y fila.
Posteriormente, el concepto de grafo quedo más claro de manera teórica pero no tanto práctica. Nos basamos en un grafo ya definido por la Dra. Elisa Schaeffer, quien en su curso de Matemáticas Discretas definía el concepto de grafo.
Así fue como programe estas 3 estructuras de datos, para más tarde ponerlas en práctica dentro de un DFS o BFS, la función que tienen estos dos es la misma en sí, pero su manera de trabajar es diferente.
Finalmente, los programas ejecutaron de manera exitosa, estos programas se encuentran de manera individual en el repositorio.