UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

FACULTAD DE CIENCIAS FÍSICO – MATEMÁTICAS

MATEMÁTICAS COMPUTACIONALES

“ESTRUCTURA DE DATOS”

Nombre: Diego Rubio Romero

Matrícula: 1738544

Carrera: Lic. En Matemáticas

Profesor: Lic. José Anastacio Hernández Saldaña

Fecha: Viernes 6 de Octubre del 2017

Lugar: Ciudad Universitaria

Introducción

Las estructuras de datos son una forma de organizar datos en una computadora para poder ser utilizado de una manera eficiente.

Las estructuras de datos manejan de manera eficiente grandes cantidades de datos para bases de datos pequeñas, hasta bases de datos grandes, etcétera.

Son clave para generar algoritmos eficientes dependiendo el tipo de método y el lenguaje que se requiera.

Existe una gran variedad de métodos de estructura de datos, pero a continuación solo se hablará de un pequeño grupo de métodos selectos en clase: Pila, Cola, Grafo, DFS, BFS.

Pila

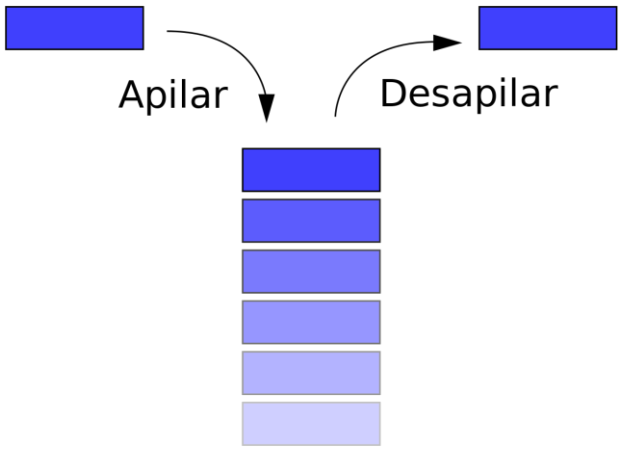
Una pila es una lista vista gráficamente como un arreglo en con una forma normalmente vertical, parecida análogamente a una pila de libros uno tras otro tras otro apilados de tal manera que sus operaciones siempre se harán en base al primer elemento del arreglo (el de la posición de mas arriba), que consiste en añadir o remover dicho primer elemento y así sobrescribirlo por otro, para seguir trabajando con éste, este modo también es llamado “Last In First Out” (último en entrar, primero en salir).

Sus dos operaciones principales consiste en apilar y retirar, respectivas a push() y pop() en python.

Por analogía con objetos cotidianos, una operación apilar equivaldría a colocar un plato sobre una pila de platos, y una operación retirar equivaldría a retirarlo.

Código fuente:

```
class Cola:
    def __init__(self):
        self.cola=[]
    def obtener(self):
        return self.cola.pop(0)
    def meter(self,e):
        self.cola.append(e)
        return len(self.cola)
    @property
    def longitud(self):
            return len(self.cola)
```



Cola

Una cola es una estructura donde los elementos nuevos llegan al final, pero el procesamiento se hace desde el primer elemento, esto se puede relacionar análogamente a la filas de persona que se forman por cierto motivo, manteniendo su procesamiento principal, el primero sale y el último entra.

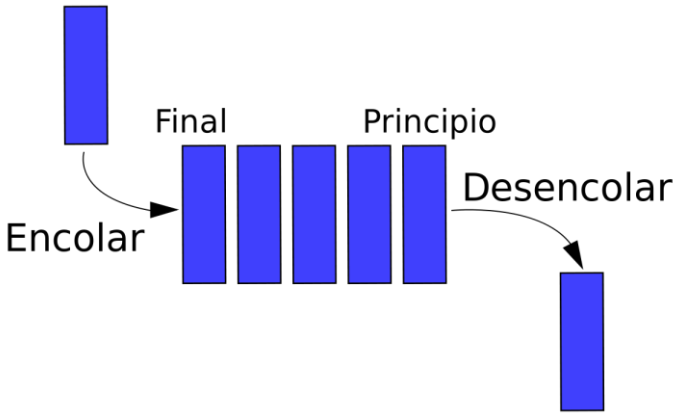
Sus dos operaciones principales consiste en apilar y retirar, respectivas a push() y pop() en python.

La particularidad de una estructura de datos de cola es el hecho de que sólo podemos acceder al primer y al último elemento de la estructura. Así mismo, los elementos sólo se pueden eliminar por el principio y sólo se pueden añadir por el final de la cola.

Ejemplos de colas en la vida real serían: personas comprando en un supermercado, esperando para entrar a ver un partido de béisbol, esperando en el cine para ver una película, una pequeña peluquería, etc. La idea esencial es que son todos líneas de espera.

Código Fuente:

```
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 longitud(self):
        return len(self.pila)
```



Grafo

Un grafo G es un par de conjuntos G=(V,E), tal que esta compuesto en base a vértices y aristas, los cuales representan gráficamente un conjunto de puntos unidos por líneas, es decir vértices unidos por aristas.

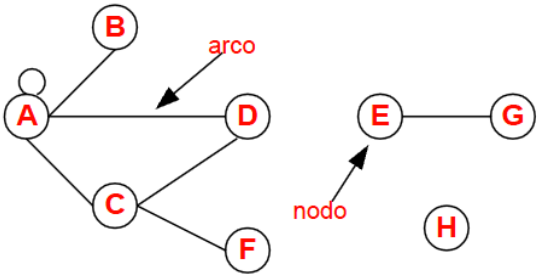
Los vértices se suelen dibujar como círculos y las aristas como líneas que les conectan uno al otro.

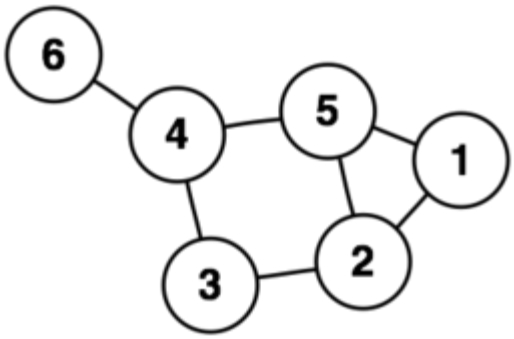
Desde un punto de vista práctico, los grafos permiten estudiar las interrelaciones entre unidades que interactúan unas con otras. Por ejemplo, una red de computadoras puede representarse y estudiarse mediante un grafo, en el cual los vértices representan terminales y las aristas representan conexiones.

Prácticamente cualquier problema puede representarse mediante un grafo, y su estudio trasciende a las diversas áreas de las ciencias exactas y las ciencias sociales.

Código Fuente:

```
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)


    @property
    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
```





Búsqueda en profundidad (Depth First Search (DFS))

Una Búsqueda en profundidad es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol de manera ordenada, pero no uniforme.

Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa, de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

Completitud: DFS es completo si y solo si usamos búsqueda basada en grafos en espacios de estado finitos, pues todos los nodos serán expandidos.

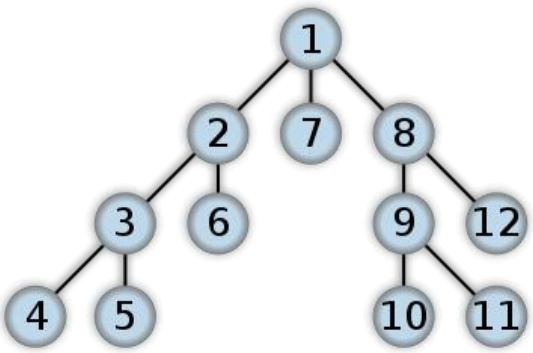
Optimalidad: DFS en ningún caso asegura la optimalidad, pues puede encontrar una solución más profunda que otra en una rama que todavía no ha sido expandida.

Complejidad temporal: en el peor caso, es $O(b^m)$, siendo b el factor de ramificación (número promedio de ramificaciones por nodo) y m la máxima profundidad del espacio de estados.

Complejidad espacial: $O(b^d)$, siendo b el factor de ramificación y d la profundidad de la solución menos costosa, pues cada nodo generado permanece en memoria, almacenándose la mayor cantidad de nodos en el nivel meta.

Código Fuente:

```
def bfs(grafo,inicio):
    visitados=[inicio]
    cola=Cola()
    cola.meter(inicio)
    while cola.longitud>0:
        actual=cola.obtener()
        vecinos=grafo.vecinos[actual]
        for i in vecinos:
            if i not in visitados:
                visitados.append(i)
                cola.meter(i)
    return visitados
```



Búsqueda en anchura (Breadth First Search (BFS))

Una Búsqueda en anchura es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo. Intuitivamente, se comienza en la raíz (eligiendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.

Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

Dado un vértice fuente s, BFS sistemáticamente explora los vértices de G para “descubrir” todos los vértices alcanzables desde s.

Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto. Llega a los nodos de distancia k, sólo tras haber llegado a todos los nodos a distancia k-1.

Código Fuente:

```
def dfs(grafo,inicio):
    visitados=[]
    pila=Pila()
    pila.meter(inicio)
    while pila.longitud>0:
        actual=pila.obtener()
        if actual in visitados:
            continue


        visitados.append(actual)
        vecinos=grafo.vecinos[actual]
        for i in vecinos:
            if i not in visitados:
                pila.meter(i)
    return visitados
```