Estructuras de Datos

Yadira Yaneth Marroquín de la Peña

Matricula: 1701504

yaz_marroquin@outlook.com

https://github.com/yadira02/1701504MatComp

6 de Septiembre del 2017

En el siguiente documento habla de las características de las estructuras de datos como Fila, Pila y Grafos. Con ellos agregaremos los algoritmos de BFS y DFS, así poder saber cómo utilizarlos con la explicación de cada uno de ellos.

1. Fila

Las colas también son llamadas FIFO (First In First Out), que quiere decir “el primero que entra es el primero que sale”. Además de ser una estructura de datos, donde se caracteriza por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pop por el otro.

```
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)


fila=Fila()
fila.meter(5)
fila.meter('fila')
```

Existen diferentes tipos de filas:

Colas simples: Se inserta por un sitio y se saca por otro, en el caso de la cola simple se inserta por el final y se saca por el principio.

Colas circulares: se considera que después del último elemento se accede de nuevo al primero.

Las filas se utilizan para formatos informativos, transportes y operaciones, otros como objetos, personas o eventos son tomados como datos que se almacenan y se aguardan mediante filas para su procedimiento.

Ventajas:

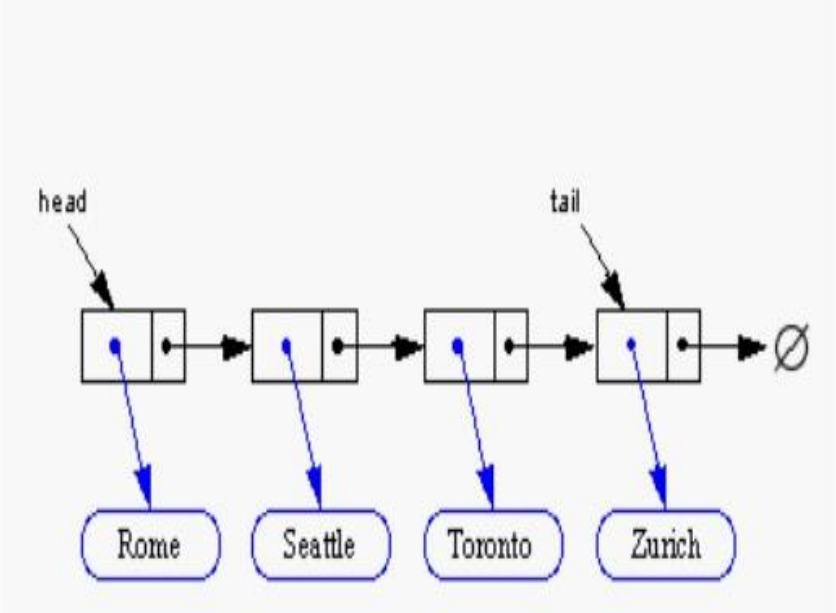
- Una fila se diferencia de una pila en que las operaciones de inserción y extracción sigue el principio de primero entrar-primero salir.
- Los elementos se pueden insertar en cualquier momento, pero solo el elemento que ha permanecido el mayor tiempo puede ser extraído.
- Los elementos se insertan al final y se extraen desde el frente.

Operaciones Básicas en Fila.

Insertar: almacena al final de fila el elemento que se recibe como parámetro.

Eliminar: saca de la fila el elemento que se encuentra al frente.

Vacía: regresa el valor indicado si la fila tiene o no elemento.



Operaciones

1.- Insertar A

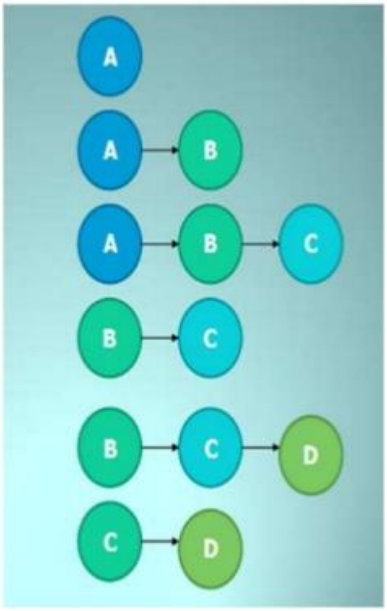
2.- Insertar B

3-.Insertar C

4-.Remover Elemento

5-. Insertar D

6.- Remover Elemento



2. Pila

Las pilas son estructuras de datos que tienes dos operaciones básicas: para insertar un elemento y para extraer un elemento. Su característica fundamental es que al extraer se obtiene siempre el último elemento que acaba de insertarse. Es una lista ordenada de elementos en la que todas las inserciones y supresiones se realizan por un mismo extremo denominado tope o cima de la 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 longitud(self):
                return len(self.pila)
>>> p=pila()
>>> p.meter(1)
```

La implementación de una pila se puede realizar mediante arreglos o con punteros. El inconveniente de la realización de una pila mediante arreglos es que su tamaño se debe especificar en tiempo. Una pila puede contener un número ilimitado de elementos y no producir nunca desbordamiento.

Operaciones de Pilas

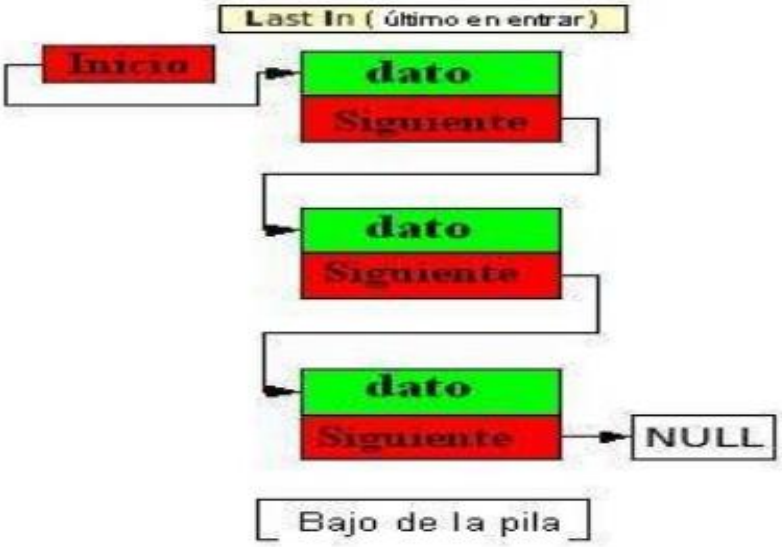
Push: introduce un elemento en la pila, también se le conoce como poner o meter.

Pop: elimina un elemento de la pila. Se le conoce como sacar o quitar.

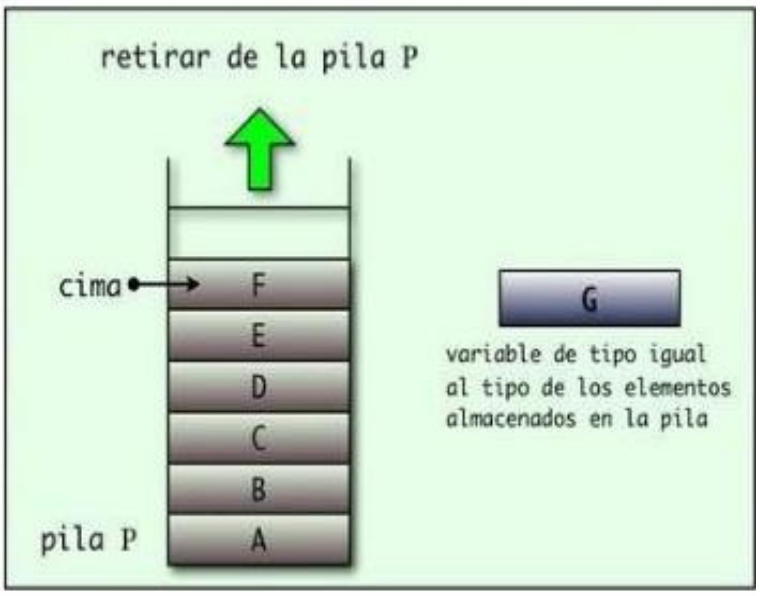
Vacía: indica si la pila es vacía o no.

Gracias a las pilas es posible el uso de la recursividad (lo veremos en detalle en el tema siguiente). La variable que llama al mismo procedimiento en el q está, habrá que guardarla así como el resto de variables de la nueva llamada, para a la vuelta de la recursividad ir sacándolas, esto es posible a la implementación de pilas.

La pila


Last In (último en entrar)
Inicio
dato
Siguiente
dato
Siguiente
dato
Siguiente
NULL
Bajo de la pila


Apilar
Desapilar
Representacion de una Pila


retirar de la pila P
cima
F
E
D
C
B
A
pila P
G
variable de tipo igual
al tipo de los elementos
almacenados en la pila

3. Grafo

Es un conjunto de puntos y un conjunto de líneas, cada una de las cuales une un punto con otro. Los puntos se llaman nodos o vértices de un grafo y las líneas se llaman aristas o arcos. Un grafo G es un conjunto en el que hay definida una relación binaria, es decir, G=(V,A) tal que V es un conjunto de objetos a los que denominaremos vértices o nodos y $A \subseteq V \times V$ es una relación binaria a cuyos elementos denominaremos arcos o aristas.

¿QUÉ ES UN NODO?

Un nodo es la unidad sobre la que se construye el árbol y puede tener cero o más nodos hijos conectados a él.

Aristas

Son las líneas con las que se unen las aristas de un grafo y con la que se construyen también caminos.

Vértices

Son los puntos o nodos con los que está conformado un grafo.

CLASIFICACION DE LOS GRAFOS

* DIRIGIDOS: Cada arco está representado por un par ordenado de vértices, de forma que representan dos arcos diferentes.

* NO DIRIGIDOS: En este el par de vértices que representa un arco no está ordenado.

TIPOS DE GRAFOS

Grafo regular: Aquel con el mismo grado en todos los vértices.

Grafo bipartito: Es aquel con cuyos vértices pueden formarse dos conjuntos disjuntos de modo que no haya adyacencias entre vértices pertenecientes al mismo conjunto

Grafo completo: Aquel con una arista entre cada par de vértices

Grafo nulo: Se dice que un grafo es nulo cuando los vértices que lo componen no están conectados, esto es, que son vértices aislados.

```
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 (búsqueda por anchura)

El funcionamiento del BFS parte del nodo agrega una fila, donde la función de la fila es entrar primero y salir primero. Supone que el recorrido se haga por niveles tomando como raíz o nodo inicial el que tiene el número 1.

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.

Implementación

- Este algoritmo de grafos la ruta más corta cuando el peso entre todos los nodos es 1.
- Cuando se requiere llegar con un movimiento de un punto a otro con el menor número de pasos, cuando se desea transformar algo un numero o cadena en otro realizando ciertas operaciones como suma producto.
- Si observan bien todo parte de un nodo inicial que será la raíz del árbol que se forma, luego ve los adyacentes a ese nodo y los agrega en un cola, los siguientes nodos a evaluar serán los adyacentes previamente insertados.

```
def BFS(self,ni):
        visitados =[]
        f=fila()
        f.meter(ni)
        while(f.longitud>0):
            na =f.obtener()
            visitados.append(na)
            vecinos = self.vecinos[na]
            for nodo in vecinos:
                if nodo not in visitados:
                    f.meter(nodo)
        return visitados
```

5. DFS

DFS permite simplificar la visión que el usuario tiene de la red y sus recursos. En esencia, DFS permite trabajar con volúmenes locales, redes compartidas, y múltiples servidores bajo un mismo sistema de archivos. En vez de pedir al usuario que recuerde los diferentes nombres de los servidores que proporcionan servicios de archivos, estos recursos pueden aparecer en el mismo sitio para el usuario, a pesar de estar en realidad separados.

VENTAJAS DE DFS

- Migración de datos simplificada: Permite mover datos de un sitio a otro sin tener que informar al usuario de la nueva localización, debido a que los usuarios no necesitan saber la localización de los datos compartidos.
- Mayor disponibilidad de los archivos: Si un cliente quiere conectar con una carpeta compartida en DFS y esa carpeta pasa a estar no disponible, DFS en ruta al usuario automáticamente a otra copia de la compartición.
- Carga de compartición: La compartición de archivos no necesariamente recae sobre un servidor, sino que se puede distribuir entre varios servidores. DFS equilibrará la carga entre múltiples copias de las carpetas compartidas.

```
def DFS(self,ni):
        visitados =[]
        f=pila()
        f.meter(ni)
        while(f.longitud>0):
            na =f.obtener()
            visitados.append(na)
            vecinos = self.vecinos[na]
            for nodo in vecinos:
                if nodo not in visitados:
                    f.meter(nodo)
        return visitados
```