Fila, Pila, Grafo, BFS y DFS

Matemáticas computacionales

Yarethzi Giselle Bazaldúa Parga

06 de octubre de 2017

Resumen

Una estructura de datos es una forma particular de organizar datos en una computadora para que pueda ser utilizado de manera eficiente. Por lo general, las estructuras de datos eficientes son clave para diseñar algoritmos eficientes. Algunos métodos formales de diseño y lenguajes de programación destacan las estructuras de datos, en lugar de los algoritmos, como el factor clave de organización en el diseño de software.

Fila

Es un tipo de dato que almacena datos, donde puedes meter datos, ver la longitud de tu fila o imprimir los elementos para visualizar lo almacenado hasta el momento.

Para programarlo me basé en un ejemplo dado en clase, a continuación se muestra el código:

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


>>> l=fila()
>>> l.meter(2)
1
>>> l.meter(5)
2
```

Pila

Se usa para almacenar datos, puedes agregar elementos, ver la longitud de tu pila, en resumen, es casi lo mismo que una fila, la diferencia de éste es que "lo último que se almacenó va a ser lo primero que se lea" como cuando apilamos platos, libros, etc.

Para hacer el código me basé en el código de fila. Se muestra a continuación:

```
>>> 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)
1
>>> p.meter(2)
2
>>> print(p.longitud)
2
>>> print(p.obtener())
2
```

Grafo:

Es un conjunto de objetos llamados vértices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto. Permiten expresar de una forma sencilla y efectiva las relaciones que se dan entre elementos de muy diversa índole.

Para realizar éste grafo tuve la ayuda de un compañero de clase, el código:

```
>>> 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
```

BFS:

Por sus siglas en inglés, significa “Búsqueda por anchura”, es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo. Lo que hace éste algoritmo es recorrer el grafo empezando de izquierda a derecha de arriba hacia abajo, ordenando así los elementos del grafo en una pila o fila, según sea el caso.

Para realizar el código me apoyé en lo visto en clase y en algunas páginas de internet:

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

DFS:

Por sus siglas en inglés, significa “Búsqueda por profundidad”, es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo de manera ordenada, pero no uniforme. Éste algoritmo recorre el grafo de arriba hacia abajo, de izquierda a derecha.

Para la elaboración de éste algoritmo, tuve la ayuda de un compañero, a continuación se muestra:

```
>>> def DFS(g,ni):
        visitados = []
        p=Pila()
        p.meter(ni)
        while(p.longitud>0):
                na =p.obtener()
                visitador.append(na)
                ln = g.vecinos[na]
                for nodo in ln:
                        if nodo not in visitados:
                                p.meter(nodo)
        return visitados
```

Para aplicar el BFS y DFS a un grafo se hace lo siguiente:

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

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

>>> def DFS(g,ni):
        visitados = []
        p=Pila()
        p.meter(ni)
        while(p.longitud>0):
                na =p.obtener()
                visitador.append(na)
                ln = g.vecinos[na]
                for nodo in ln:
                        if nodo not in visitados:
                                p.meter(nodo)
        return visitados

g= Grafo()
g.conecta('a','b',3)
g.conecta('a','c',3)
g.conecta('a','d',1)
g.conecta('a','d',2)
g.conecta('b','e',2)
g.conecta('b','f',2)
g.conecta('c','g',2)
g.conecta('e','h',2)
g.conecta('x','z',1)

print(BFS(g,'a'))
print(BFS(g,'x'))
print(DFS(g,'a'))
```