



Alumno: Rivera Navarrete Miguel Ángel amado.

Profesor: Blancas Ruiz José Luis.

Ciclo escolar: 19-2.

Carrera: Ing. Sistemas Computacionales.

Tema: Practica 3 tema con el Stack y con la recursividad.

**Tema con la función Stack.**

**DFSs.**

from Nodos import Nodo ''' Esta parte es para importar la libreria o la clase de Nodo '''

from Estados import Estado ''' Esta parte es para importar la libreria o la clase de Estado '''

#import sys

class DFSrecursivo():

''' Esta clase ejecuta el DFS recursivo '''

def \_\_init\_\_(self): ''' Aqui definimos el metodo o constructor de la clase '''

self.encontrado = False ''' self nos ayuda a que se le de un valor y ademas es una instancia '''

def busqueda(self):

''' Este método ejecutará una búsqueda recursiva'''

estadoInicial = Estado() ''' Esta es un metodo '''

print(estadoInicial) ''' imprimimos la variable el cual se esta generando un metodo en otra clase se manda llamar '''

nodoRaiz = Nodo(estadoInicial)

self.DFS(nodoRaiz)

def DFS(self, nodo):

''' Crear el árbol de búsqueda '''

if not self.encontrado:

print('-- procesar -- ',nodo.estado.lugar

#Checar si hemos alcanzado el estado meta

if nodo.estado.checaMeta(): ''' se declara un metodo de busqueda para buscar el recorrido '''

print('Meta Alcanzada')

self.encontrado = True

else:

# encuentra a los sucesores del estado actual

estadosHijos = nodo.estado.funcionSucesora()

for estadoHijo in estadosHijos:

nodoHijo = Nodo(Estado(estadoHijo))

nodo.agregaHijos(nodoHijo)

self.DFS(nodoHijo)

dfs = DFSrecursivo() ''' se ejecuta el metodo '''

dfs.busqueda()

**DFS.**

from Nodos import Nodo # importar las clase Nodo para generar nuestra búsqueda

from Estados import Estado # importar las clase Estado para generar nuestra búsqueda

import sys

''' El uso del stack es para simular la política FIFO o UEPS, donde vamos a viajar sobre el último nodo

generado a profundidad'''

def DFS(): # este es la clase DFS

stack = [] #el stack es para viajar sobre los nodos

estadoInicial = Estado()

raiz = Nodo(estadoInicial)

print("Nodo Raiz ---> ",raiz.estado.lugar)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print("Atributos Raiz ---> ", raiz.\_\_dict\_\_)

stack.append(raiz)

''' Imprimir el contenido del stack, a través de desplegar los atributos del objeto stack'''

for x in stack:

print('Valor del Stack ', x.estado.\_\_dict\_\_)

print('--------------------------------------------------------------')

contador = 0

while len(stack) > 0: ''' El ciclo while que nos va ayudar a buscar si el recorrido del arbol es correcta '''

nodoActual = stack.pop()

if nodoActual.estado.checaMeta():

print("Meta Alcanzada", nodoActual.estado.lugar)

break

estadosHijos = nodoActual.estado.funcionSucesora()

#estamos construyendo un nuevo objeto, tipo clase nodo

# clase estado, con el componente estado hijo segregado

for estadoHijo in estadosHijos:

nodoHijo = Nodo(Estado(estadoHijo))

#y en en el stack hijos, agrega a los hijos

nodoActual.agregaHijos(nodoHijo)

for index in range(len(nodoActual.hijos)-1,-1,-1): ''' el ciclo for se ocupa para saber que valores hay en el con ayuda de range len para ver su tamaño y se manda llamar el metodo o la variable que se creó anteriormente '''

stack.append(nodoActual.hijos[index])

for x in stack:

sys.stdout.write('{},'.format(x.estado.\_\_dict\_\_,))

sys.stdout.flush()

print('==')

contador = contador + 1

#print(contador)

# Imprimir el árbol

print("----------------------")

raiz.imprimeArbol()

DFS()

**Estado.**

from grafico1 import conexion

class Estado:

'''

Esta clase obtiene la información del estado para nuestra aplicación de búsqueda.

'''

def \_\_init\_\_(self, lugar = None):

if lugar == None:

#create initial state

self.lugar = self.obtenEstadoInicial()

else:

self.lugar = lugar

def obtenEstadoInicial(self):

"""

Este método regresa como estado inicial el definido como estado inicial del grafico que se este usando.

"""

estadoInicial = "Parada Autobus"

return estadoInicial

def funcionSucesora(self):

''' Método debe regresar la función sucesora, de la lista serán los valores de la llave de acceso del

Diccionario'''

#print("NODO SUCESOR ::::> ",conexion[self.lugar])

return conexion[self.lugar]

def checaMeta(self):

"""

Este método checa si en la ruta está el estado meta.

"""

return self.lugar == "Laboratorios Computo"

**Nodo.**

class Nodo:

'''

Esta el nodo en la búsqueda de árbol/gráfico

'''

def \_\_init\_\_(self, estado):

"""

Constructor

"""

self.estado = estado

self.hijos = []

self.padre = None

def agregaHijos(self, nodoHijo):

"""

Este método agrega un nodo debajo de otro nodo.

"""

self.hijos.append(nodoHijo)

nodoHijo.padre = self

def imprimeArbol(self):

"""

Este método imprime un árbol.

"""

print(" - " , self.estado.lugar)

for hijo in self.hijos:

hijo.imprimeArbol()

**Grafico.**

''' aqui se hacen las busquedas de como se va a construir el arbaol que recorrido es el que va a realizar al momento de mandarlo llamar '''

#Conexiones entre lugares

conexion = {} ''' este es un arreglo vasio '''

conexion["Parada Autobus"] = {"Libreria"} ''' los corchetes son para definir la relacion que llevara y las llaves son para saber hacia donde va apuntar esa variable '''

conexion["Libreria"] = {"Estacionamiento", "Centro Estudiantil"}

conexion["Estacionamiento"] = {"Edificio Administrativo", "Tienda"}

conexion["Edificio Administrativo"] = {"Cafeteria"}

conexion["Centro Estudiantil"] = {"Tienda" , "Teatro"}

conexion["Tienda"] = {"Cafeteria"}

conexion["Cafeteria"] = {"Laboratorios Computo"}

#conexion["Laboratorios Computo"] = {"Cafeteria"}

conexion["Teatro"] = {"Gimnasio"}

conexion["Gimnasio"] = {"Tienda"}

**Tema con recursividad.**

**DFSs.**

from Nodos import Nodo ''' Esta parte es para importar la libreria o la clase de Nodo '''

from Estados import Estado ''' Esta parte es para importar la libreria o la clase de Estado '''

#import sys

class DFSrecursivo():

''' Esta clase ejecuta el DFS recursivo '''

def \_\_init\_\_(self): ''' Aqui definimos el metodo o constructor de la clase '''

self.encontrado = False ''' self nos ayuda a que se le de un valor y ademas es una instancia '''

def busqueda(self):

''' Este método ejecutará una búsqueda recursiva'''

estadoInicial = Estado() ''' Esta es un metodo '''

print(estadoInicial) ''' imprimimos la variable el cual se esta generando un metodo en otra clase se manda llamar '''

nodoRaiz = Nodo(estadoInicial)

self.DFS(nodoRaiz)

def DFS(self, nodo):

''' Crear el árbol de búsqueda '''

if not self.encontrado:

print('-- procesar -- ',nodo.estado.lugar

#Checar si hemos alcanzado el estado meta

if nodo.estado.checaMeta(): ''' se declara un metodo de busqueda para buscar el recorrido '''

print('Meta Alcanzada')

self.encontrado = True

else:

# encuentra a los sucesores del estado actual

estadosHijos = nodo.estado.funcionSucesora()

for estadoHijo in estadosHijos:

nodoHijo = Nodo(Estado(estadoHijo))

nodo.agregaHijos(nodoHijo)

self.DFS(nodoHijo)

dfs = DFSrecursivo() ''' se ejecuta el metodo '''

dfs.busqueda()

**DFS.**

from Nodos import Nodo # importar las clase Nodo para generar nuestra búsqueda

from Estados import Estado # importar las clase Estado para generar nuestra búsqueda

import sys

''' El uso del stack es para simular la política FIFO o UEPS, donde vamos a viajar sobre el último nodo

generado a profundidad'''

def DFS(): # este es la clase DFS

stack = [] #el stack es para viajar sobre los nodos

estadoInicial = Estado()

raiz = Nodo(estadoInicial)

print("Nodo Raiz ---> ",raiz.estado.lugar)

print('\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*')

print("Atributos Raiz ---> ", raiz.\_\_dict\_\_)

stack.append(raiz)

''' Imprimir el contenido del stack, a través de desplegar los atributos del objeto stack'''

for x in stack:

print('Valor del Stack ', x.estado.\_\_dict\_\_)

print('--------------------------------------------------------------')

contador = 0

while len(stack) > 0: ''' El ciclo while que nos va ayudar a buscar si el recorrido del arbol es correcta '''

nodoActual = stack.pop()

if nodoActual.estado.checaMeta():

print("Meta Alcanzada", nodoActual.estado.lugar)

break

estadosHijos = nodoActual.estado.funcionSucesora()

#estamos construyendo un nuevo objeto, tipo clase nodo

# clase estado, con el componente estado hijo segregado

for estadoHijo in estadosHijos:

nodoHijo = Nodo(Estado(estadoHijo))

#y en en el stack hijos, agrega a los hijos

nodoActual.agregaHijos(nodoHijo)

for index in range(len(nodoActual.hijos)-1,-1,-1): ''' el ciclo for se ocupa para saber que valores hay en el con ayuda de range len para ver su tamaño y se manda llamar el metodo o la variable que se creó anteriormente '''

stack.append(nodoActual.hijos[index])

for x in stack:

sys.stdout.write('{},'.format(x.estado.\_\_dict\_\_,))

sys.stdout.flush()

print('==')

contador = contador + 1

#print(contador)

# Imprimir el árbol

print("----------------------")

raiz.imprimeArbol()

DFS()

**Estado.**

from grafico1 import conexion

class Estado:

'''

Esta clase obtiene la información del estado para nuestra aplicación de búsqueda.

'''

def \_\_init\_\_(self, lugar = None):

if lugar == None:

#create initial state

self.lugar = self.obtenEstadoInicial()

else:

self.lugar = lugar

def obtenEstadoInicial(self):

"""

Este método regresa como estado inicial el definido como estado inicial del grafico que se este usando.

"""

estadoInicial = "Parada Autobus"

return estadoInicial

def funcionSucesora(self):

''' Método debe regresar la función sucesora, de la lista serán los valores de la llave de acceso del

Diccionario'''

#print("NODO SUCESOR ::::> ",conexion[self.lugar])

return conexion[self.lugar]

def checaMeta(self):

"""

Este método checa si en la ruta está el estado meta.

"""

return self.lugar == "Laboratorios Computo"

**Nodo.**

class Nodo:

'''

Esta el nodo en la búsqueda de árbol/gráfico

'''

def \_\_init\_\_(self, estado):

"""

Constructor

"""

self.estado = estado

self.hijos = []

self.padre = None

def agregaHijos(self, nodoHijo):

"""

Este método agrega un nodo debajo de otro nodo.

"""

self.hijos.append(nodoHijo)

nodoHijo.padre = self

def imprimeArbol(self):

"""

Este método imprime un árbol.

"""

print(" - " , self.estado.lugar)

for hijo in self.hijos:

hijo.imprimeArbol()

**Grafico.**

''' aqui se hacen las busquedas de como se va a construir el arbaol que recorrido es el que va a realizar al momento de mandarlo llamar '''

#Conexiones entre lugares

conexion = {} ''' este es un arreglo vasio '''

conexion["Parada Autobus"] = {"Libreria"} ''' los corchetes son para definir la relacion que llevara y las llaves son para saber hacia donde va apuntar esa variable '''

conexion["Libreria"] = {"Estacionamiento", "Centro Estudiantil"}

conexion["Estacionamiento"] = {"Edificio Administrativo", "Tienda"}

conexion["Edificio Administrativo"] = {"Cafeteria"}

conexion["Centro Estudiantil"] = {"Tienda" , "Teatro"}

conexion["Tienda"] = {"Cafeteria"}

conexion["Cafeteria"] = {"Laboratorios Computo"}

#conexion["Laboratorios Computo"] = {"Cafeteria"}

conexion["Teatro"] = {"Gimnasio"}

conexion["Gimnasio"] = {"Tienda"}

**DFSrs.**

from Nodos import Nodo #importamos las librerías de nodo.

from Estados import Estado #importamos las librerías de estados.

class DFSrecursivo(): #se crea la clase de DFSrecursivo.

''' Esta clase ejecuta el DFS recursivo '''

def \_\_init\_\_(self):

self.encontrado = False

def busqueda(self): #se crea un método donde le dirigimos self para que haga la búsqueda.

''' Este método ejecutará una búsqueda recursiva'''

estadoInicial = Estado() #aqui se posiciona en el nodo inicial para hacer el recorrido.

print(estadoInicial)

nodoRaiz = Nodo(estadoInicial)

self.DFS(nodoRaiz)

def DFS(self, nodo): #se crea un método el cual se le dirige hacia el nodo para realizar la búsqueda.

''' Crear el árbol de búsqueda 'para ver si lo encuentra o no''

if not self.encontrado:

print('-- procesar -- ',nodo.estado.lugar)

#Checar si hemos alcanzado el estado meta

if nodo.estado.checaMeta():

print('Meta Alcanzada')

self.encontrado = True

else:

# encuentra a los sucesores del estado actual

estadosHijos = nodo.estado.funcionSucesora()

for estadoHijo in estadosHijos:

nodoHijo = Nodo(Estado(estadoHijo))

nodo.agregaHijos(nodoHijo)

self.DFS(nodoHijo)

dfs = DFSrecursivo()

dfs.busqueda()