Universidad Autónoma
del Estado de México.

Facultad de Ingeniera.

Implementación de un árbol en python.

Materia: Programación Avanzada

Alumna: Diana Quintana Gamboa

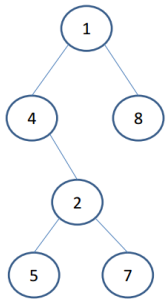
Profesor: Elfego Gutiérrez Ocampo

Fecha: 11 de Noviembre del 2016

¿Cómo implementar un árbol binario en python?

Un árbol es una estructura jerárquica sobre un conjunto de objetos llamados nodos. La jerarquía entre los nodos se establece por medio de conectores llamados ramas. Existe un nodo especial llamado raíz o nodo padre. Representan una estructura no-lineal y dinámica de datos más importantes.

Tienen las siguientes propiedades:



- ♣ Tienen un nodo llamado “raíz”.
- ♣ Todos los nodos, excepto la raíz, tienen una sola line a de entrada.
- ♣ Existe solo una única ruta de la raíz a todos los demás nodos.
- ♣ En caso de tener otra ramificación se le considera nodo padre y nodo hijo

Esta es una implementación para la creación de árboles que no forzosamente tienen que estar balanceados.

```
def __init__(self,valor):
        self.valor = valor
        self.izquierda = None
       self.derecha = None

#Método para agregar nodos a la izquierda del árbol, no importando que nodo
#del árbol queremos como padre de este
def AgregaIzquierda(self,padre,dato):
        if self.valor != padre:
        if self.izquierda != None:
                self.izquierda.AgregaIzquierda(padre,dato)
        if self.derecha!=None:
                self.derecha.AgregaIzquierda(padre,dato)
       else:
                self.izquierda=Arbol(dato)

#Metodo para agregar nodos a la derecha, no importando que nodo del arbol
#queremos como padre de este
def AgregaDerecha(self,padre,dato):
         if self.valor != padre:
        if self.izquierda != None:
                self.izquierda.AgregaDerecha(padre,dato)
        if self.derecha!=None:
                self.derecha.AgregaDerecha(padre,dato)
        else:
               self.derecha = Arbol(dato)
```

```
#Metodo que nos permite buscar un elemento en forma recursiva
def BuscaNodo(self,dato):
        if self.valor != dato:
        if self.izquierda!=None:
                return self.izquierda.BuscaNodo(dato)
        if self.derecha!=None:
                return self.derecha.BuscaNodo(dato)
        else:
                return self.valor

#Metodo para impresion del arbol comenzando Primero Izquierda usando
recursividad
def ImprimeArbolIzq(self):
        if self.valor!=None:
                print self.valor
        if self.izquierda!=None:
                self.izquierda.ImprimeArbolIzq()
        if self.derecha!=None:
                self.derecha.ImprimeArbolIzq()

#Metodo para impresion del arbol comenzando primero por la derecha usando
recursividad
def ImprimeArbolDer(self):
        if self.valor!=None:
                print self.valor
        if self.derecha!=None:
                self.derecha.ImprimeArbolDer()
        if self.izquierda!=None:
                self.izquierda.ImprimeArbolDer()
```

Ejemplos

1). El problema del laberinto

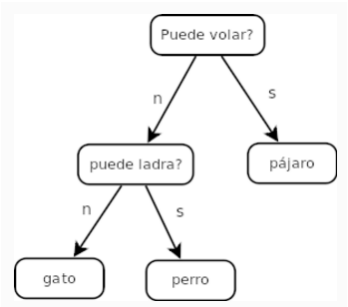
```
stringLab = "00000000000000\n" + \
            "01111111111110\n" + \
            "00000000000010\n" + \
            "01111111111110\n" + \
            "01111111111110\n" + \
            "01111111111110\n" + \
            "00000000003110"

def creaLaberinto(stringLab) :
    """ Crea un laberinto a partir de una tira de entrada.
        Entradas:
             stringLab : tira que contiene el diseño del
                         laberinto.
        Salidas:
             Laberinto representado por una matriz, tal que
             la entrada i,j contiene: 0 - si la casilla está
             libre, 1 - si hay pared, 3 - posición en donde
             está el queso.
        Restricciones:
             Todas las entradas de la tira son 0, 1 o 3. Las
             filas se representan por un cambio de línea.
             No hay líneas vacías.
    """

    lista = stringLab.split()
    lista = [ x[:-1] if x[-1] == "\n" else x for x in lista]
    lista = [[int(ch) for ch in x] for x in lista]
    return lista
def impLab(laberinto):
    """ Imprime un laberinto.
        Entradas:
             laberinto : laberinto a imprimir.
        Salidas:
             Ninguna.
        Restricciones:
             El laberinto está representado por listas de
             listas, y es una representación consistente. """

    for x in laberinto:
        for y in x:
            print(y, end= "")
        print()

laberinto = creaLaberinto(stringLab)
impLab(laberinto)

def recorrido(i, j):
    """ Dado un laberinto en donde se ubica un queso,
        retorna en una lista de pares ordenados (x,y)
        que indican el camino desde una posición inicial
        (i,j) hasta la posición en que se encuentra el
        queso.
        Entradas:
             (i, j) : posición inicial a partir de donde
                      se realizará la búsqueda de un camino
                      hasta la posición del queso.
        Salidas:
             Lista con las casillas, expresadas como pares
             ordenados, que llevan desde la posición inicial
             hasta la posición en que se encuentra el queso.
             Si no existe un camino retorna la lista vacía.
    """

    if laberinto[i][j] == 3:
        return [(i, j)]

    if laberinto[i][j] == 1:
        return []

    laberinto[i][j] = -1

    if i > 0 and laberinto[i - 1][j] in [0, 3]:
        camino = recorrido(i - 1, j)
        if camino: return [(i, j)] + camino

    if j < len(laberinto[i]) - 1 and laberinto[i][j + 1] in [0, 3]
        camino = recorrido(i, j + 1)
        if camino: return [(i, j)] + camino

    if i < len(laberinto) - 1 and laberinto[i + 1][j] in [0, 3]:
        camino = recorrido(i + 1, j)
        if camino: return [(i, j)] + camino

    if j > 0 and laberinto[i][j - 1] in [0, 3]:
        camino = recorrido(i, j - 1)
        if camino: return [(i, j)] + camino

    return []

   for x in recorrido(6,13) : print(x)
```

2). Suma

```
class Arbol:
    def __init__(self, carga=None, izq=None, der=None):
        self.carga = carga
        self.izquierda = izq
        self.derecha = der

    def __str__(self):
        return str(self.carga)


def suma(arbol):
    if arbol == None: return 0
    return suma(arbol.izquierda) + suma(arbol.derecha) + arbol.carga

arbol = Arbol(5, Arbol(4), Arbol(3))
print suma(arbol)
```

3).

```
class Arbol:
    def __init__(self, carga=None, izq=None, der=None):
        self.carga = carga
        self.izquierda = izq
        self.derecha = der

    def __str__(self):
        return str(self.carga)

# -----------------------------------------------------------

# Funciones
# -----------------------------------------------------------

def si(preg):
    from string import lower
    resp = lower(raw_input(preg))
    return (resp[0] == 's')

def main():
    bucle = True
    raiz = Arbol("pajaro")
    while bucle:
        if not si("Estas pensando en un animal? "): break

        arbol = raiz
        while arbol.izquierda != None:
            if si(arbol.carga + "? "):
                arbol = arbol.izquierda
            else:
                arbol = arbol.derecha

        #adivinar
        animal = arbol.carga
        if si("Es un " + animal + "? "):
            print "Soy el más grande!"
            continue

        #obtener informacion
        nuevo = raw_input("Qué animal era? ")
        info = raw_input("Qué diferencia a un " + animal + " de un " + nuevo + "? ")
        indicador = "Si el animal fuera un " + animal + " cual seria la respuesta? "
        arbol.carga = info
        if si(indicador):
            arbol.izquierda = Arbol(animal)
            arbol.derecha = Arbol(nuevo)

        else:
            arbol.derecha = Arbol(animal)
            arbol.izquierda = Arbol(nuevo)

    return 0

if __name__ == '__main__':
    main()
```



4). Recorridos de un árbol

```
def ejecutarPreOrden(arbol, funcion):
    if (arbol != None):
        funcion(arbol.elemento)
        ejecutarPreOrden(arbol.izquierda, funcion)
        ejecutarPreOrden(arbol.derecha, funcion)

def ejecutarInOrden(arbol, funcion):
    if (arbol != None):
        ejecutarInOrden(arbol.izquierda, funcion)
        funcion(arbol.elemento)
        ejecutarInOrden(arbol.derecha, funcion)

def ejecutarPostOrden(arbol, funcion):
    if (arbol != None):
        ejecutarPostOrden(arbol.izquierda, funcion)
        ejecutarPostOrden(arbol.derecha, funcion)
        funcion(arbol.elemento)
```

5). Este recorrido recibe el nombre de búsqueda por primero en profundidad.

```
def backTracking(v):
    assert v es una lista v[0], v[1], ..., v[i]
    assert v[k] es un punto de la solución

    if vector es una solución:
        return vector

    for vp in posibles(v[i]):
        if v + [vp] es un vector acceptable:
            sol = backTracking(v + [vp])
            if sol != []:
                return sol

    return []
```

Referencias.

https://sites.google.com/site/programacioniiuno/temario/unidad-5---grafos/rboles-binarios

http://www.linuxsc.net/the-prog/480-creacion-de-un-arbol-de-busqueda-en-python

http://pensandocomoprogramador.blogspot.mx/2012/07/c7-arboles-en-python.html

http://www.solveet.com/exercises/Arboles-Binarios/175/solution-1257

http://www.genbetadev.com/paradigmas-de-programacion/crear-un-adivinador