Árbol N ario

Son una extensión de los árboles binarios en donde cada nodo puede tener un número variable de hijos, esto los hace ideales para generar estructuras jerárquicas complejas. Se utilizan por ejemplo para representar sistemas de archivos. Al igual que un árbol binario es una jerarquía la cual está compuesta por nodos, hay un nodo principal (el que contiene a todos los nodos y es el punto de acceso) en este libro lo llamaremos raíz. Y cada hijo del nodo principal puede tener múltiples hijos y así sucesivamente.

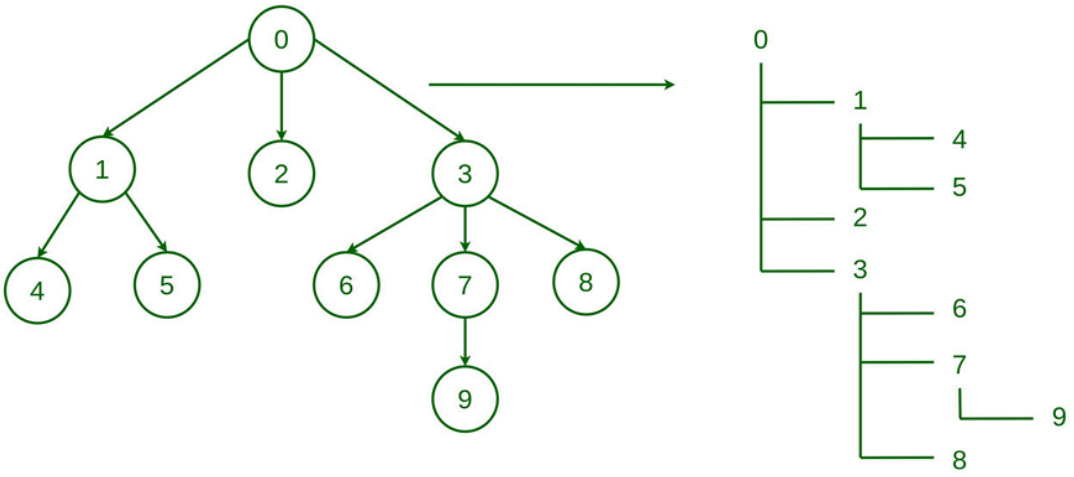

FIG 00: representación de la estructura de un árbol N ario.

Por qué existen los árboles N arios

- Representación de jerarquías: Los sistemas de carpeta y subcarpetas se pueden representar con árboles N arios debido a que un Nodo puede tener múltiples hijos.

- Organización en niveles de jerarquía: En ciertos tipos de bases de datos NoSQL los datos se organizan en niveles de jerarquía los cuales pueden ser más complejos que una estructura binaria simple.

- Toma de decisiones: En videojuegos de estrategia cada movimiento puede llevar a múltiples estados, lo cual se puede modelar de una mejor manera usando árboles N arios.

Un árbol n ario no es más que una colección de nodos, los cuales contienen un arreglo que puede contener múltiples nodos. Por ello cada nodo contiene 2 elementos:

FIG 01: representación de un nodo al lado de su respectivo código.

Para poder utilizar esto necesitamos crear algo que se llama "Árbol N-ario"

Un Árbol es una colección de nodos que nos va a permitir guardar la información, dicha clase contiene un apuntador (el principal es la raíz)... si usted observa la siguiente figura usted podrá notar cómo funciona:

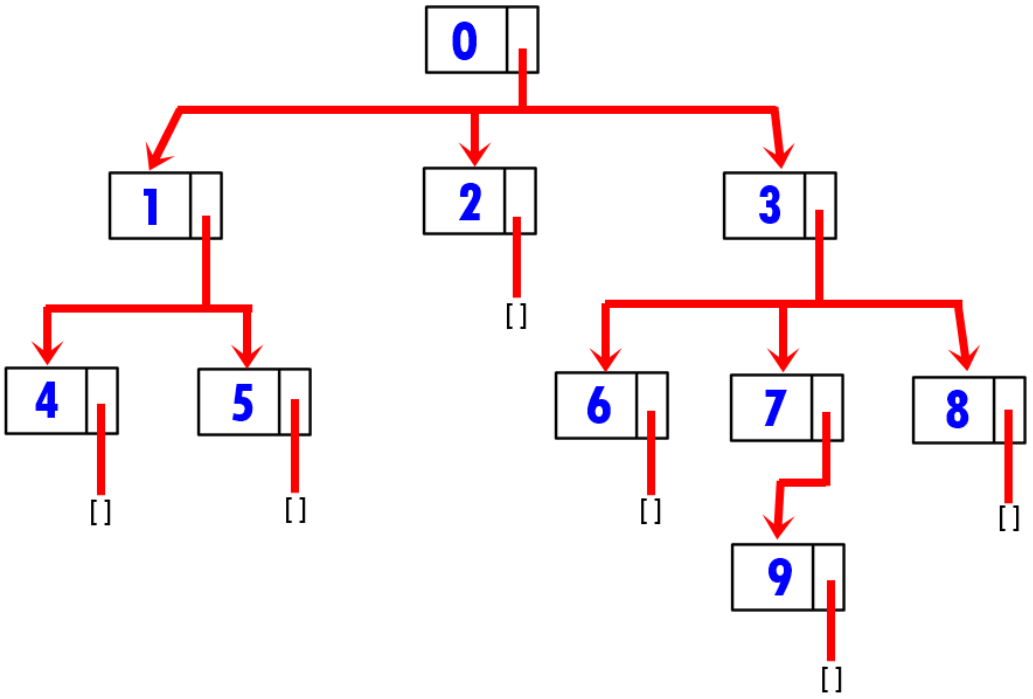

FIG 02: ejemplo visual de un árbol N ario.

Los árboles n-arios necesitan de los siguientes atributos y métodos para funcionar:

- Se necesita un pivote el cual es el encargado de moverse para leer/escribir la información.
- Se necesita un método para insertar un valor en su respectivo padre.
- Se necesita un método para buscar un nodo por valor de data.
- Se necesita un método para borrar un nodo y todos sus descendientes.
- Se necesita un método para recorrer el árbol en preorden.
- Se necesita un método para recorrer el árbol en postorden.

Que es el pivote

El pivote no es más que un nodo el cual vamos a utilizar para poder movernos entre los Nodos de ese árbol (Moverse entre las ramas) con este movimiento nosotros vamos a poder:

- Agregar, Eliminar, Buscar y Modificar los datos de un nodo.

El pivote es el encargado de poder ser el punto de acceso para movernos en el árbol, sin él no tendríamos una forma de recorrer el árbol.

```
class NTree:
    def __init__(self):
        self.root = None
```

Advertencia: El nodo principal inicia en None debido a que no existe DATA para construir el primer nodo.

Como se agregan los elementos a un árbol

para lograr la inserción de elementos en un árbol N-ario necesitamos 3 cosas:

- Un padre.
- Un dato.
- Usar la recursividad.

Para ello tenemos que partir de 5 casos base:

- Caso 1.0 La raíz está vacía y se envía un padre y un dato: Lo único que tenemos que hacer es crear un nodo principal con la información del padre y luego guardar el dato en la respectiva jerarquía del padre.

```
if self.root == None:
    if father != None:
        self.root = Node(father)

    if father != None and data != None:
        self._addData(self.root, father, data)
```

- Caso 1.1 la raíz está vacía y solo se envía el padre: Lo único que tenemos que hacer es almacenar la respectiva información del padre.

```
if self.root == None:
    if father != None:
        self.root = Node(father)
```

- Caso 1.2 la raíz está vacía y solo se envia el dato: Lo único que tenemos que hacer es almacenar la respectiva información como si fuera el padre root:

```
def addData(self, father, data):
    if self.root == None:
        if father != None: ···

        if father != None and data != None: ···

        if father == None and data != None:
            self.root = Node(data)
```

- caso 2: El árbol ya contiene información y sabemos cual es el padre: En ese caso lo único que tenemos que hacer es agregar un nuevo nodo en el array del padre:

```
def addData(self, father, data):
    if self.root == None: ···
    else:
        self._addData(self.root, father, data)
def _addData(self, pivot, father, data):
    if pivot.data == father:
        pivot.children.append(Node(data))
```

- caso 3: El árbol ya contiene información y NO sabemos cual es el padre:En ese caso lo que tenemos que hacer es encontrar recursivamente el padre (Aunque necesitamos un for de sus hijos) y cuando encontremos al padre que estamos buscando volver a realizar el caso 2:

```python
def addData(self, father, data):
    if self.root == None:...
    else:
        self._addData(self.root, father, data)
def _addData(self, pivot, father, data):
    if pivot.data == father:
        pivot.children.append(Node(data))
    else:
        for i in pivot.children:
            self._addData(i, father, data)
```

Buscar Nodo

para poder buscar un nodo tenemos que hacer uso de la recursividad mezclada con ciclos, osea tenemos que buscar un padre entre todos sus hijos y luego retorna el Nodo correspondiente según sea su valor

```python
def searchByData(self, data):
    if self.root != None:
        return self._searchByData(self.root, data)

    return None
def _searchByData(self, pivot, data):
    if pivot.data == data:
        return pivot

    for i in pivot.children:
        result = self._searchByData(i, data)

        if result != None:
            return result

    return None
```

Borrar un Nodo y todos sus descendientes

lamentablemente no existen operaciones como en los árboles binarios que nos permitan copiar los valores hijos… debido a que una de las principales reglas de los ärboles N-arios es respetar la jerarquía, por ello lo que tenemos que hacer es buscar a el padre y luego eliminar a ese hijo con todos sus respectivos descendientes:

```
def deleteNode(self, father, data):
    if self.root != None and father != None and data != None:
        self._deleteNode(self.root, father, data)
def _deleteNode(self, pivot, father, data):
    if pivot.data == father:
        pivot.children = [node for node in pivot.children if node.data != data]
    else:
        for i in pivot.children:
            self._deleteNode(i, father, data)
```

Cuando el nodo es encontrado lo que se hace es que a su padre se le hace una copia de todos los nodos excepto el nodo que se quiere eliminar.

Recorridos

Al igual que en los árboles binarios se tienen varios recorridos según el tiempo de acceso a la data, para el caso de los árboles N arios tenemos de 2 tipos:

- Pre order: primero se imprime y luego se explora entre los descendientes.
- Post order: primero se explora y luego se imprime

```
def viewPreOrder(self):
    self._viewPreOrder(self.root)
def _viewPreOrder(self, pivot):
    if pivot != None:
        print(pivot.data)

        for i in pivot.children:
            self._viewPreOrder(i)
```

```
def viewPostOrder(self):
    self._viewPostOrder(self.root)
def _viewPostOrder(self, pivot):
    if pivot != None:
        for i in pivot.children:
            self._viewPostOrder(i)

        print(pivot.data)
```