Listas circulares

Es una variación de las listas simplemente enlazada donde el último nodo apunta al primer nodo (formando un círculo) . Esta estructura es útil en aplicaciones que requieren un ciclo continuo.

Una lista circular no es más que una colección de nodos donde cada nodo contiene estos 2 elementos:

- Dato.
- Nodo siguiente.

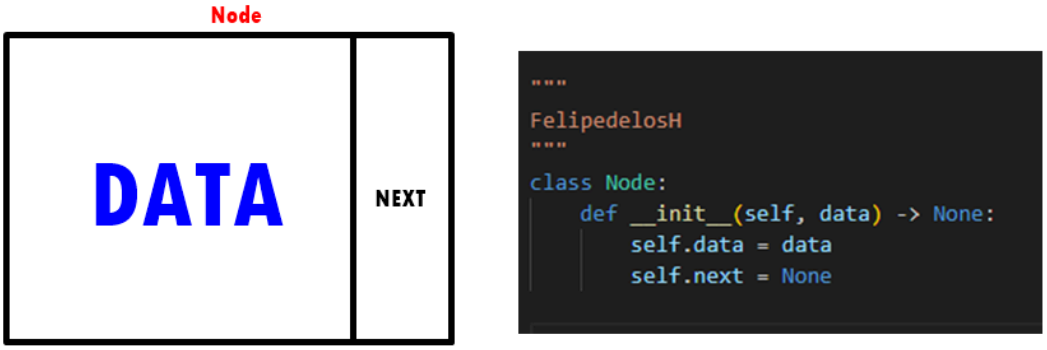

FIG 00: representación de un nodo al lado de su respectivo código.

Para poder utilizar esto necesitamos crear algo que se llama “lista circular”

Una lista circular es una colección de nodos que nos va a permitir guardar la información pero con la peculiaridad que el último nodo siempre apunta al pivote, dicha clase contiene un apuntador y varios datos… si observa la siguiente figura usted podrá notar cómo funciona:

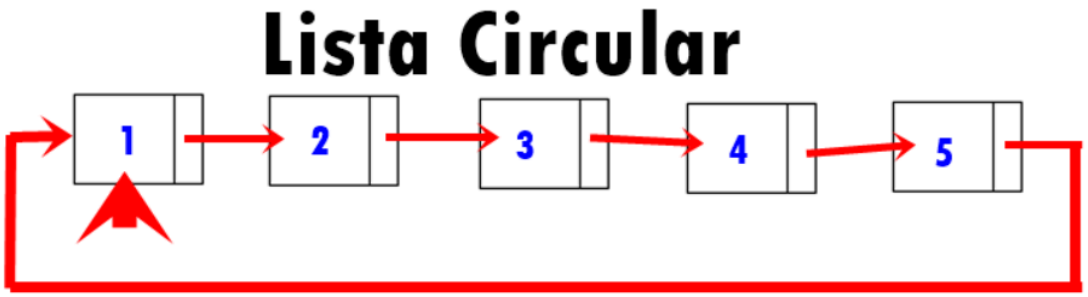

FIG 01: representación gráfica de una lista simple circular.

Las listas necesitan los siguientes atributos y métodos para poder funcionar:

- Se necesita un pivote el cual es el encargado de moverse para leer/escribir la información.
- Se necesita un método para agregar.
- Se necesita un método para mostrar toda la lista.

Advertencia: con esos pocos métodos nuestra lista es funcional.

Que es el pivote

El pivote no es más que una instancia del nodo, pero no podemos empezar la lista instanciando un nodo sin una data (o al menos no es recomendable) el pivote es la cabeza del nodo y se encarga de moverse para recorrer la lista y poder agregar elementos.

El pivote/cabezal es el encargado de ser el punto de acceso a los datos de la lista sin él no existiría una forma de recorrer la lista.

```
class CircularLinkedList:
    def __init__(self) -> None:
        self.pivot = None
```

Advertencia: el pivote inicia en None dado a que cuando la lista es creada no posee datos.

Como se agregan elementos a la lista

Yo he decidido hacerlo de manera recursiva aunque también es posible hacerlo de manera iterativa. para agregar a la lista se tiene que partir de 2 casos base:

- Caso 1: la lista está vacía: La solución es simple se declara un nuevo nodo con la información a almacenar y luego el pivote se asigna a el nuevo nodo luego procederemos a linkear el siguiente con el mismo pivote formando así un ciclo:

```
new_node = Node(data)
if self.pivot == None:
    self.pivot = new_node
    self.pivot.next = self.pivot
```

- Caso 2: La lista no está vacía: En este caso lo que hacemos es sacarle una copia al pivote y luego recorremos esa copia con un while para situarnos (un nodo antes de dónde apunta el pivote) se inserta en esa posición y nuevamente hacemos que el nuevo nodo apunte hacia el pivote.

```
_copy = self.pivot

while _copy.next != self.pivot:
    _copy = _copy.next

_copy.next = new_node
new_node.next = self.pivot
```

Advertencia: se saca copia del pivote por precaución, para evitar borrar el original.

Métodos que le dan valor agregado a la lista:

- Se necesita un método para contar el total de elementos de la lista.
- Se necesita un método para verificar si existe un dato en la lista.
- Se necesita un método para actualizar un valor de la lista.
- Se necesita un método para eliminar un elemento.
- Se necesita un método para saber si la lista está vacía.
- Se necesita un método para obtener un dato en una posición x.

Cómo contar el total de elementos

Lo que necesitamos es recorrer la lista (con la condición de detenerse en el pivote/cabeza) y utilizar un contador para retornar el valor.

```python
def count(self):
    if self.pivot == None:
        return 0

    _counter = 1
    _copy = self.pivot

    while _copy.next != self.pivot:
        _counter = _counter + 1
        _copy = _copy.next

    return _counter
```

Verificar si existe un dato en la lista

Lo que hacemos es buscar el elemento mediante un ciclo while (con la condición de detenerse en el pivote/cabeza) y se retorna si el elemento fue encontrado:

```python
def isDataInList(self, data):
    if self.pivot == None:
        return False

    _copy = self.pivot

    while _copy.next != self.pivot:
        if _copy.data == data:
            return True

        _copy = _copy.next

    return False
```

actualizar un valor de la lista

```
def updateValue(self, index, data):
    _count = self.count()

    if index >= 0 and index < _count:
        _copy = self.pivot
        _counter = 0
        while _counter < index:
            _copy = _copy.next
            _counter = _counter + 1

        # UPDATE
        _copy.data =  data
```

para eliminar un elemento

```
def deleteValue(self, data):
    if self.pivot != None:
        _copy = self.pivot

        # Case 0: the data is in a head
        if _copy.data == data:
            # The list only have single element
            if _copy.next == self.pivot:
                self.pivot = None
            # The list have more elements
            else:
                while _copy.next != self.pivot:
                    _copy = _copy.next

                self.pivot = self.pivot.next
                _copy.next = _copy.next.next
        # Case 1: the data it'nt in a heath
        else:
            _previous = None
            while _copy.next != self.pivot:
                _previous = _copy
                _copy = _copy.next

                if _copy.data == data:
                    _previous.next = _copy.next
```