Universidad Nacional de Entre Ríos

Facultad de Ingeniería

Carrera:

Tecnicatura Universitaria en Procesamiento y Explotación de Datos

Materia:

Algoritmos y Estructura de Datos
Tercera Cohorte

Equipo de cátedra:

Jordán F. Insfrán, Javier Eduardo Diaz Zamboni, Diana Vertiz del Valle, Belén Ferster, Bruno Breggia

Trabajo práctico N° 1

Alumnos:

Carrozo Felipe, Juan Pablo Gonzalez

Fecha:

30/9/2022

Ejercicio 1

Empezamos este ejercicio creando la clase `Nodo`. En el método mágico `__init__` (inicializador) se definen las partes esenciales de un nodo: referencia al dato, al nodo siguiente y al nodo anterior. Asimismo, este método tiene como parámetro `datoInicial`, el cual se requerirá cuando se cree el objeto `Nodo`. En las propiedades del nodo (getters y setters), se definen `dato`, `siguiente` y `anterior`, las cuales fueron definidas en el inicializador.

```
class Nodo:

    def __init__(self,datoInicial):
        self.dato = datoInicial
        self.siguiente = None
        self.anterior = None
```

Luego, creamos la clase `ListaDobleEnlazada`. En su método mágico `__init__` se definen las referencias de la lista: la `cola` (inicio), la `cabeza` (fin) y el `tamaño` (longitud). La `cabeza` y la `cola` apuntan a `None` y `tamanio` a cero.

```
class ListaDobleEnlazada:

    def __init__(self):
        self.cabeza = None
        self.cola = None
        self._tamanio = 0
```

Entre los otros métodos mágicos que implementamos, están `__iter__` y `__str__`. El primero nos permite iterar sobre la lista doblemente enlazada, es decir, usar sentencias for y/o while y que el retorno no sea "`TypeError: 'ListaDobleEnlazada' object is not iterable`", sino que recorra el largo de la lista, elemento por elemento, o nodo por nodo. El método mágico `__str__` nos permite imprimir por consola los resultados; es decir, que cuando se llama a la función `print(objeto)`, este no muestre por consola la ubicación en memoria del objeto.

Para saber si una lista está vacía, simplemente creamos un método `esta_vacia()`, la cual retorna `True` si el tamaño de la lista es igual a cero, de lo contrario, retorna `False` (datos de tipo booleano).

Si queremos agregar un elemento al principio de una lista inicializada con la clase ListaDobleEnlazada podemos utilizar el método `agregar`, especificando el item que deseamos incorporar a la lista. En pocas palabras, los nodos que ya existen se moverán un lugar adelante y el nuevo se ubicará en la primera posición de la lista. Esta tarea se lleva a cabo a través de la modificación de la referencia antecesora que tiene el primer nodo (ya existente), que ahora debe apuntar al nuevo nodo, el cuál ocupará el lugar de la cabeza. Por último, este método modificará el tamaño de la lista incrementandola en 1 unidad.

Posiblemente queramos añadir objetos al final de nuestra ListaDobleEnlazada, y para eso, creamos el método `anexar` el cuál va a modificar también las referencias, pero esta vez, del último nodo existente en la fila con respecto a su predecesor que será el nodo que queremos anexar. Entonces: la lista incrementará su tamaño, el nodo nuevo se ubicará en el final de la lista -también llamada “cola”-, y el nodo que antes se ubicaba en este lugar ahora cambia su referencia con respecto al siguiente, de: None (tierra) a el nuevo nodo.

Para el método `insertar(posicion, ítem)` se necesita de una posición en la lista, y un elemento a insertar, el cual hará que los nodos que están a sus lados se muevan un lugar para atrás y otro para delante respectivamente. Si esto pasa y no se retorna un `IndexError` (por motivos de posición fuera de rango), el tamaño incrementará en una unidad más.

En el caso de querer sacar un item en particular de la lista, la clase tiene un método llamado `extraer(posición)`, el cuál pedirá una posición lógica para extraer un objeto de la lista, es decir, no menor a -1 ni mayor al tamaño de la lista y modificará las referencias del nodo seleccionado o de su predecesor y antecesor (en el caso de que no esté ubicado en la cabeza o en la cola) con el objetivo de que se pierda la referencia del mismo, pero no sin antes guardar su contenido en una variable auxiliar/temporal para que muestre al usuario el nodo que se elimina de la lista. Este método en vez de incrementar el tamaño como hacían los anteriores, lo disminuye.

El método `copiar` no requiere ningún argumento. Este crea una nueva lista doblemente enlazada vacía. Luego itera con una sentencia `for` sobre la lista que se desea copiar y va escribiendo sobre la nueva lista cada nodo.

La función `concatenar (lista)` recibe una lista como parámetro. La cola del objeto y la cabeza de la lista pasada por parámetro se unen para formar una sola lista. No es necesario crear una nueva lista.

Una variante para concatenar listas es a través del método mágico `__add__`, el cuál va a permitir unir dos listas utilizando el operador de suma (+).

Si lo que se quiere es invertir el orden los elementos de una lista, la clase tiene un método llamado `invertir`. Este, siempre que haya más de un elemento en la lista, va a modificar las referencias de los nodos para que cada uno cambie su referencia al anterior pasando a ser la de su siguiente.

Por último, el método `ordenar`. En nuestro caso, el método de ordenamiento implementado fue el de burbuja. Este, toma de a dos elementos y los compara: si el primero es más chico que el segundo, está ordenado y el cambio no se produce. De lo contrario, invierte las posiciones de estos dos elementos. Esta forma de ordenamiento se produce tantas veces como sea necesario, hasta que la lista esté totalmente ordenada. De esta forma, primero se encuentra el número mayor, luego el que le sigue, y así sucesivamente. En cada pasada se encuentra el número decrecientemente.

Graficamos el orden de complejidad del tamaño de la lista en función del tiempo que tarda en terminar de ejecutar el programa.