Laboratorio Nro. 3: Implementación de Listas Enlazadas

Alejandro Murillo González
Universidad Eafit
Medellín, Colombia
amurillog@eafit.edu.co

Juan Pablo Vidal Correa
Universidad Eafit
Medellín, Colombia
Jpvidalc@ eafit.edu.co

3) *3) Simulacro de preguntas de sustentación de Proyectos*

3.1 3.1 Teniendo en cuenta lo anterior, verifiquen, utilizando JUnit, que todos los tests escritos en el numeral 1.2 pasan:

Pruebas Unit hechas en python:

```
C:\Users\user\Anaconda2\python.exe "C:\Program Files\JetBrains\PyCharm Community Edition 2017.2.1\helpers\pycharm\_jb_nosete
Testing started at 6:58 PM ...
Launching Nosetest with arguments C:\Program Files\JetBrains\PyCharm Community Edition 2017.2.1\helpers\pycharm\_jb_nosetest
test_empty_linked_list (Laboratorio 4.LinkedList_UnitTests) ... ok
test_insertion (Laboratorio 4.LinkedList_UnitTests) ... ok
test_non_empty_linked_list (Laboratorio 4.LinkedList_UnitTests) ... ok

----------------------------------------------------------------------
Ran 3 tests in 0.000s

OK
...
----------------------------------------------------------------------
Ran 3 tests in 0.002s

OK

Process finished with exit code 0
```

3.2 Expliquen con sus propias palabras cómo funciona la implementación del ejercicio 2.1:

Se implementa una Pila que interprete y ejecute la serie de comandos, para esto se deben realizar métodos que inserten los elementos en la pila, y que los saque para que se pueda efectuar cada operación, pero además se debe tener en cuenta que algunas operaciones no estarán en orden, por lo que al sacar algunos elementos o bloques, para insertar un elemento u organizar, se debe volver a meter estos bloques, para que se pueda seguir la línea de

operaciones, por lo que se deben hacer métodos que saquen todos los bloques y los inserte según las ordenes.

3.3 Calculen la complejidad del ejercicio realizado en el numeral 2.1

La complejidad de este ejercicio es O(n), ya que eligiendo el peor de los casos solo tiene un ciclo.

Nota: no hay ciclos anidados.

Código:

```
class RoboticArm:


    def __init__(self):


        self.number_of_blocks = int(input("Number of blocks: "))


        self.block_world = [[i] for i in range(self.number_of_blocks)]


        next_command = input("Commands:\n")
        while next_command != "quit":              //n
            command_pieces = next_command.split()


            a = int(command_pieces[1])
            b = int(command_pieces[3])


            if command_pieces[0] == "move":     //C*n
                if command_pieces[2] == "onto":  //C*n
                    self.move_onto(a, b)
                elif command_pieces[2] == "over": //C*n
                    self.move_over(a, b)
            elif command_pieces[0] == "pile":     //C*n
```

```
            if command_pieces[2] == "onto":    //C*n
               self.pile_onto(a, b)
            elif command_pieces[2] == "over":  //C*n
               self.pile_over(a, b)


         next_command = input()


      self.quit()



def move_onto(self, a, b):

      a_block, b_block = self.block_assignment(a, b)


      a_index = a_block.index(a)
      b_index = b_block.index(b)


      while len(a_block) - 1 > a_index:      //C*a
         over_block = a_block.pop()
         self.block_world[over_block].append(over_block)


      while len(b_block) - 1 > b_index:      //C*b
         over_block = b_block.pop()
         self.block_world[over_block].append(over_block)


      b_block.append(a_block.pop())


def move_over(self, a, b):
```

```
    a_block, b_block = self.block_assignment(a, b)

    a_index = a_block.index(a)

    while len(a_block) - 1 > a_index:          //C*a
        over_block = a_block.pop()
        self.block_world[over_block].append(over_block)

    b_block.append(a_block.pop())

def pile_onto(self, a, b):

    a_block, b_block = self.block_assignment(a, b)

    a_index = a_block.index(a)
    blocks_over_a = a_block[a_index:]
    b_index = b_block.index(b)

    while len(a_block) - 1 > a_index:
        a_block.pop()

    while len(b_block) - 1 > b_index:       //C*b
        over_block = b_block.pop()
        self.block_world[over_block].append(over_block)

    b_block += blocks_over_a

def pile_over(self, a, b):
```

```
    a_block, b_block = self.block_assignment(a, b)


    a_index = a_block.index(a)
    blocks_over_a = []


    while len(a_block) > a_index:            //C*a
        blocks_over_a = [a_block.pop()] + blocks_over_a


    b_block += blocks_over_a


def quit(self):
    """
    Stops the robot's actions, and
    prints the final arrangement
    of the blocks.

    :return: Void
    """


    for index, block in enumerate(self.block_world):       //C*n
        line = str(index) + ": "
        if len(block) > 0:                          //C*n
            line += str(block)[1:-1].replace(",", "")
        print(line)


def block_assignment(self, a, b):


    a_block = b_block = None
```

DOCENTE MAURICIO TORO BERMÚDEZ
Teléfono: (+57) (4) 261 95 00 Ext. 9473. Oficina: 19 - 627
Correo: mtorobe@eafit.edu.co

```
for block in self.block_world:              //C*n
   if a in block:                        //C*n
      a_block = block
   if b in block:                         //C*n
      b_block = block
   if a_block != None and b_block != None:     //C*n
      break


return a_block, b_block
```

```
robot = RoboticArm()
```

T(n) = O(C x a + C x b +C x n) == T(n) es O(n)

3.4 Expliquen con sus palabras las variables (qué es 'n', qué es 'm', etc.) del cálculo de complejidad del numeral 3.3

Las variables "n", "m" y/o "y" son las variables de entrada, esto con el objetivo de facilitar los cálculos de complejidad.

4) Simulacro de Parcial

1.

a) lista.size()

b) lista.add(auxiliar.pop());

2.

a) auxiliar1.size() > 0, auxiliar2.size() > 0

b) personas.offer(edad)

3. C

5) Lectura recomendada (opcional)

a) Resumen:

Pilas y Colas:

Pilas:
Una pila es una estructura de datos que permite el acceso a un único elemento de datos, el cual es siempre el último elemento insertado. Cuando se retira este elemento de la lista, se puede acceder al siguiente elemento, el cual se convertiría en el último elemento insertado, y de esa forma se puede acceder a cada elemento.

Esta estructura utiliza los siguientes métodos para manipular los elementos:

Push: permite ingresar un elemento a la pila.

Pop: saca el elemento que se encuentra al comienzo de la pila

Peek: inspecciona el elemento que se encuentra al comienzo de la pila, sin sacarlo.

Colas:
Es una estructura de datos que se asemeja a una Pila ya que solo se puede inspeccionar un elemento a la vez, pero se diferencian debido a que en las Colas el primer elemento insertado es el primero en salir.

Esta estructura utiliza los siguientes métodos:

Insert: inserta un elemento a la cola.

Remove: elimina un elemento de la cola.

Peek: cumple la misma función que en las Pilas.
Existen otros tipos de Colas:

Cola doble: Es una cola con doble final, en donde se puede elegir que camino toma el elemento.

Cola circular: Los elementos removidos vuelven al inicio de la cola.

6) Trabajo en Equipo y Progreso Gradual (Opcional)

a) Actas de reunión:

Integrante	Fecha	Hecho	Haciendo	Por Hacer
Murillo	5/10/2017	Implemento linked list en python	Pruebas Unit	
Vidal	5/10/2017	Implemento código en línea (2.1)	Calculando complejidad	leer lectura opcional y resumen
Murillo	6/10/2017	Código numeral 1.3	Organizar texto- informe de laboratorio	Agregar partes opcionales de Vidal
Vidal	8/09/2017	implemento resumen	Simulacro de parcial	Organizar texto- Informe de laboratorio