



CICLO 1

[FORMACIÓN POR CICLOS]

Fundamentos de **PROGRAMACIÓN**



Ingeni@
Soluciones TIC



UNIVERSIDAD
DE ANTIOQUIA

Facultad de Ingeniería

Lectura

LISTAS

ligadas



Introducción

Hasta ahora, en nuestro curso hemos trabajado colecciones de datos representándolas en vectores (arreglos de una dimensión). Dicha representación presenta algunos problemas, como desperdicio de memoria o memoria insuficiente, además de que las operaciones de inserción y borrado se efectúan con algoritmos cuyo orden de magnitud es lineal, $O(n)$, lo cual se considera ineficiente en tareas que tienen alta frecuencia de ejecución. Por tanto, veremos aquí una nueva forma para representar colecciones de datos de tal manera que no aparezcan los problemas de memoria insuficiente o desperdicio de memoria, es decir, que los programas utilicen exactamente la memoria que necesitan, y que los algoritmos para los proceso de inserción y borrado sean eficientes, es decir, que tengan orden de magnitud $O(1)$.

Manejo estático de la memoria.

Para entrar a hablar del manejo dinámico de memoria repasemos brevemente lo que es el manejo estático de la memoria, es decir, los arreglos. Tratemos el caso de un vector, según la clase definida previamente:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
b	d	f	H	I	m									

Figura 1

En el vector de la figura 1 tenemos un conjunto de datos ordenados ascendenteamente.

Las operaciones básicas que se realizan sobre ese conjunto de datos son insertar un dato y borrar un dato. Analicemos cada una de ellas.

Si se desea insertar el dato **g** en el vector de la figura 1, el proceso a seguir es:

1. Buscar en cuál posición insertarlo (método **buscarDondeInsertar(d)**).
2. Insertarlo en la posición correspondiente (método **insertar(d, i)**).

Del primer paso se obtiene que la posición en la cual hay que insertar la **g** es la posición 4 del vector. El segundo paso deberá mover los datos desde la posición 4 hasta la posición 6, una posición hacia la derecha y luego asignar a la posición 4 el dato **g**.

Si el dato a insertar hubiera sido la letra **a**, hubiéramos tenido que mover todos los datos del vector.

Generalizando, se considera el peor de los casos, que es cuando hay que insertar el dato al principio del vector. Hay que mover **n** datos en el vector, lo cual implica que el algoritmo de inserción tendrá orden de magnitud **O(n)**.

Si deseamos borrar un dato de un vector, los pasos a seguir son:

1. Buscar en cuál posición se halla el dato a borrar (método **buscarDatos(d)**).
2. Borrar el dato del vector (método **borrarDatosEnPosicion(i)**).

En caso de que el dato a borrar fuera la letra **f**, el primer paso me retorna 3, o sea, la posición en la cual se halla la letra **f**. El segundo paso moverá los datos desde la posición 4 hasta la 6 una posición hacia la izquierda.

Si el dato a borrar hubiera estado en la posición 1 del vector y el vector tiene **n** datos, entonces habrá que mover **n - 1** datos hacia la izquierda, y el orden de magnitud de dicho algoritmo es **O(n)**.

De lo expuesto anteriormente, los algoritmos de inserción y borrado tienen orden de magnitud **O(n)**, y esto en el manejo de grandes volúmenes de información, con operaciones de alta frecuencia, como son insertar y borrar, se considera ineficiente.

Por tanto, se ha buscado una forma alterna de representación en la cual las operaciones de inserción y borrado sean eficientes, es decir, tengan orden de magnitud **O(1)**.

Concepto.

Consideremos los siguientes dos vectores: **DATO** Y **LIGA**.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
DATO	d	h	b		m	f		i						
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
LIGA	6	8	1		+	2		5						

Figura 2

En el vector **DATO** almacenamos los datos en posiciones aleatorias. Físicamente los datos se hallan en desorden. Nos interesa tenerlos ordenados lógicamente. El símbolo **+** representa nulo.

Para ello debemos conocer en cuál posición del vector se halla el primer dato. En nuestro ejemplo, el primer dato se halla en la posición 3 del vector. Utilizaremos una variable, llamémosla **primero**, cuyo valor es 3. Es decir, el hecho de que primero valga 3 significa que el primer dato se halla en la posición 3 del vector **DATO**.

Nos interesa saber en cuál posición se halla el siguiente dato, para lo cual utilizamos la posición 3 del vector **LIGA**.

El siguiente dato, que es la d, se halla en la posición 1 del vector **DATO**; por tanto, en la posición 3 del vector **LIGA** tendremos un 1.

El hecho de que en la posición 3 del vector **LIGA** haya un 1 significa que el siguiente dato se halla en la posición 1 del vector **DATO**.



Para conocer el siguiente a la d utilizamos la posición 1 del vector **LIGA**. Allí encontramos un 6, lo que significa que el siguiente dato a la d se encuentra en la posición 6 del vector **DATO**.

En general, para un dato que se halle en la posición i del vector **DATO**, la correspondiente posición i del vector **LIGA** indica en cuál posición del vector **DATO** se halla el siguiente dato.

Los textos de computadores suelen presentar la situación descrita anteriormente, de la siguiente forma:

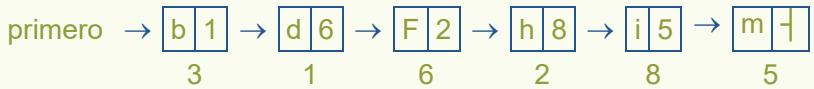


Figura 3

donde cada cuadrito contiene dos campos: uno para el dato y otro para la liga. Técnicamente, cada cuadrito se denomina **nodo**.

Clase nodo simple.

Como nuestro objetivo es trabajar programación orientada a objetos, y para ello utilizamos clases con sus respectivos métodos, comencemos definiendo la clase en la cual manipulemos el nodo que acabamos de definir.

A dicha clase la denominaremos **nodoSimple**.

```
class nodoSimple:  
    def __init__(self, d = None):  
        self.dato = d  
        self.liga = None  
  
    def asignarDato(self, d):  
        self.dato = d  
  
    def asignarLiga(self, x):  
        self.liga = x  
  
    def retornarDato(self):  
        return self.dato  
  
    def retornarLiga(self):  
        return self.liga
```



Como podrá observar, los métodos correspondientes a esta clase son bastante simples. Solo consisten en algoritmos para modificar los datos de un nodo (dato y liga) y los algoritmos para acceder a dichos datos.

Habiendo definido la clase **nodoSimple**, en la siguiente lectura veremos la clase lista simplemente **ligada (LSL)**, que se construye con objetos de la clase **nodoSimple**, y en la que las operaciones de insertar y borrar será con algoritmos cuyo orden de magnitud es **O(1)**.

En nuestra clase hemos definido un campo de dato llamado **d**. En este campo se podrá almacenar lo que el usuario necesite: un entero, un número con decimales, un carácter, un string, un registro, un vector, una matriz, etc.

Con base en objetos de la clase **nodoSimple** podemos trabajar colecciones de datos, representando cada uno de los datos en un nodo y teniendo conectados todos los nodos correspondientes a dicha colección. Trataremos ahora la manipulación de estos conjuntos de nodos conectados, conocidos como listas simplemente ligadas.

Clase Lista Simplemente Ligada.

Es un conjunto de nodos conectados cuyo elemento básico son objetos de la clase **nodoSimple**. Con el fin de poder operar sobre este conjunto de nodos es necesario conocer el primer nodo del conjunto de nodos que están conectados y en muchas situaciones el último nodo del conjunto. Con base en esto vamos a definir una clase llamada **LSL**, la cual tendrá dos datos privados de la clase **nodoSimple**, que llamaremos primero y ultimo: primero apuntará hacia el primer nodo de la lista y ultimo apuntará hacia el último nodo de la lista. Además, definiremos las operaciones que podremos efectuar sobre objetos de dicha clase.

Se recomienda al estudiante que, por el momento, se concentre en las operaciones definidas (las resaltadas en amarillo) y en lo que hace cada una de ellas. Teniendo claro ese conocimiento, podrá definir listas y trabajar con ellas.

Teniendo superado ese proceso, podrá entrar a analizar y racionalizar los algoritmos correspondientes a dichos métodos.

Como dato importante, tenga presente que los atributos definidos en el constructor (primero y ultimo) son objetos de la clase nodoSimple.

```
class LSL:  
    def __init__(self):          #Constructor  
        self.primero = None  
        self.ultimo = None  
  
    def primerNodo(self):  
        return self.primero  
  
    def ultimoNodo(self):  
        return self.ultimo  
  
    def esVacia(self):  
        return self.primero == None  
  
    def finDeRecorrido(self, p):  
        return p == None  
  
    def recorrerLista(self):  
        p = self.primerNodo()  
        while not self.finDeRecorrido(p):  
            print(p.retornarDatos(), end = ", ")  
            p = p.retornarLiga()  
  
    def agregarDatos(self, d):  
        x = nodoSimple(d)  
        p = self.primerNodo()  
        if p == None:  
            self.primero = x  
            self.ultimo = x  
        else:  
            self.ultimo.liga = x  
            self.ultimo = x  
  
    def buscarDondeInsertar(self, d):  
        p = self.primerNodo()  
        y = None  
        while not self.finDeRecorrido(p) and p.retornarDatos() <  
d:  
        y = p  
        p = p.retornarLiga()  
return y
```



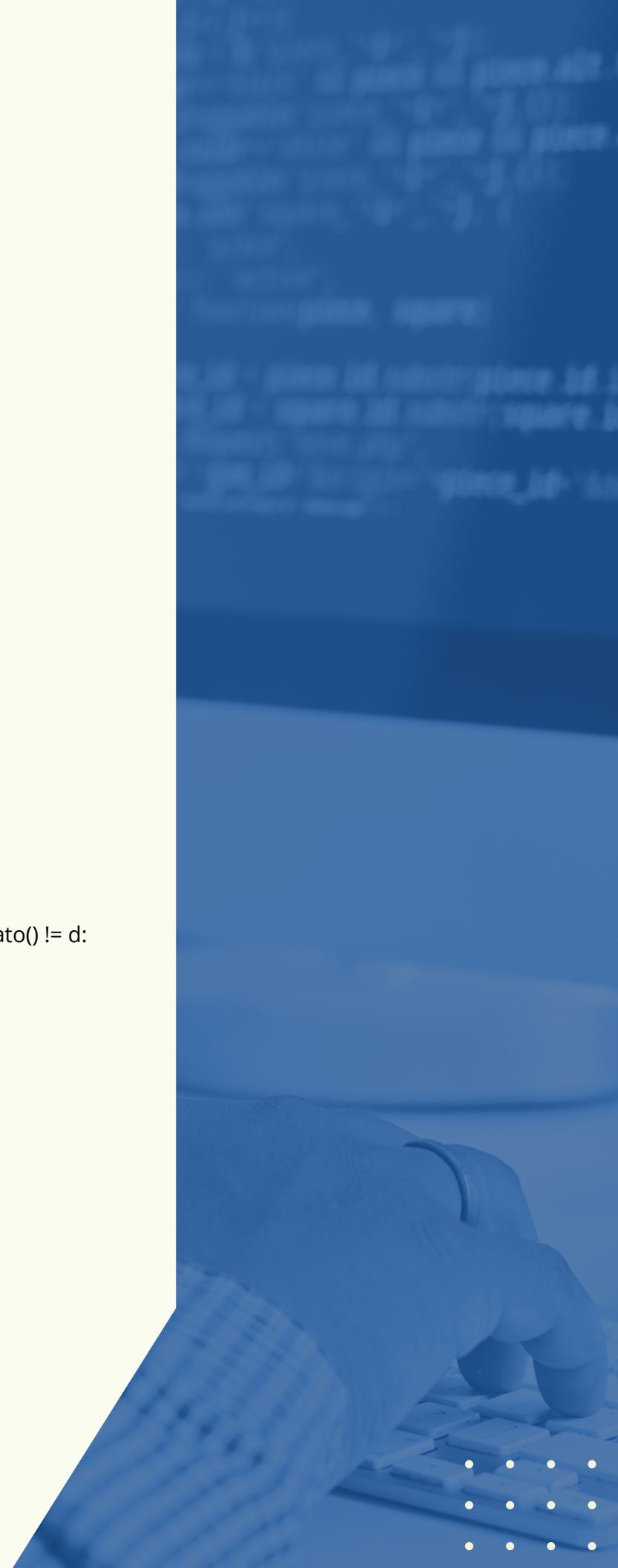
```
def insertar(self, d, y):
    x = nodoSimple(d)
    self.conectar(x, y)

def conectar(self, x, y):
    if y == None:
        if self.primero == None:
            self.ultimo = x
        else:
            x.asignarLiga(self.primero)
            self.primero = x
        return
    x.asignarLiga(y.retornarLiga())
    y.asignarLiga(x)
    if y == self.ultimo:
        self.ultimo = x

def longitud(self):
    p = self.primerNodo()
    n = 0
    while not self.finDeRecorrido(p):
        n = n + 1
        p = p.retornarLiga()
    return n

def buscarData(self, d, y):
    x = self.primerNodo()
    while not self.finDeRecorrido(x) and x.retornarDato() != d:
        y.asignarDato(x)
        x = x.retornarLiga()
    return x

def borrar(self, x, y = None):
    if x == None:
        print("Dato no está en la lista")
        return
    if y == None:
        if x != self.primero:
            print("Falta el anterior del dato a borrar")
            return
    else:
        y = y.retornarDato()
        self.desconectar(x, y)
```



```

def desconectar(self, x, y):
    if y == None:
        self.primer = x.retornarLiga()
    if self.esVacia():
        self.ultimo = None
    else:
        y.asignarLiga(x.retornarLiga())
        if x == self.ultimo:
            self.ultimo = y

```

Expliquemos brevemente cada uno de los métodos definidos. Comencemos con el constructor. Cuando se ejecuta el constructor, lo único que se hace es crear una nueva instancia de la clase LSL con sus correspondientes datos privados en **None**. Ejemplo: si tenemos estas instrucciones:

```
a = LSL()
b = LSL()
```

lo que se obtiene es:

a → **primero = None**
ultimo = None

b → **primero = None**
ultimo = None

La función **esVacia()** retorna verdadero si la lista que invoca el método está vacía, falso de lo contrario. Una lista está vacía cuando la variable **primero** es **None**.

Para explicar qué es lo que hace cada uno de los métodos definidos en la clase **LSL**, consideremos el siguiente objeto, perteneciente a la clase **LSL**, y que llamamos **a**:

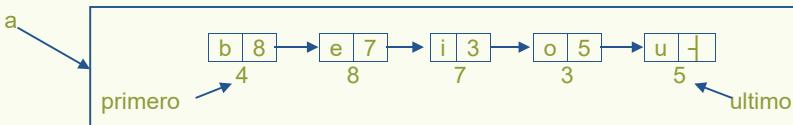


Figura 4



La función **primerNodo()** retorna el nodo que está de primero en la lista. Si efectuamos la instrucción **p = a.primerNodo()**, entonces **p** quedará valiendo 4.

La función **ultimoNodo()** retorna el nodo que está de último en la lista. Si ejecutamos la instrucción **p = a.ultimoNodo()**, entonces **p** quedará valiendo 5.

La función **finDeRecorrido(p)** retorna verdadero si el nodo **p** enviado como parámetro es **None**, falso de lo contrario.

La función **anterior(x)** retorna el nodo anterior al nodo enviado como parámetro. Si tenemos que **x = 7** y ejecutamos la instrucción **p = a.anterior(x)**, entonces **p** quedará valiendo 8.

El método **recorrerLista()**, como su nombre lo dice, simplemente recorre y escribe los datos de una lista simplemente ligada. Si ejecutamos la instrucción **a.recorrerLista()**, el resultado que se obtiene es la escritura; b, e, i, o, u.

La función **buscarDondeInsertar(d)** retorna el nodo a continuación del cual se debe insertar un nuevo nodo con dato **d** en una lista simplemente ligada en la cual los datos están ordenados ascendente y deben continuar cumpliendo esta característica después de insertarlo. Veamos algunos ejemplos:

y = a.buscarDondeInsertar('f') entonces **y** queda valiendo 8.
y = a.buscarDondeInsertar('z') entonces **y** queda valiendo 5
y = a.buscarDondeInsertar('a') entonces **y** queda valiendo None.

El método **insertar(d, y)** consigue un nuevo **nodoSimple**, lo carga con el dato **d**, e invoca el método conectar con el fin de conectar el nuevo nodo (llamémoslo **x**) a continuación del nodo **y**. Si ejecutamos las siguientes instrucciones:

```
d = 'f'  
y = a.buscarDondeInsertar(d)  
a.insertar(d, y)
```

el objeto **a** quedará así:

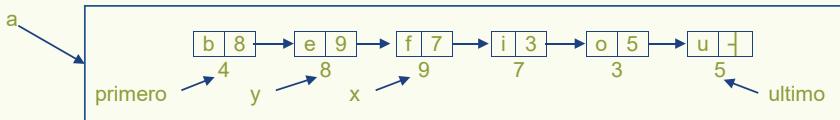


Figura 5

El método conectar simplemente conecta el nodo **x** a continuación del nodo **y**, tal como se ve en la figura 5.

La función **buscarDatos(d, y)**, como su nombre lo dice, busca el dato **d** en la lista que invoca el método: si lo encuentra, retorna el nodo en el cual lo encontró, de lo contrario retorna **None**. En el campo de dato del parámetro **y**, retorna el nodo anterior al nodo en el cual encontró el dato **d**. Algunos ejemplos, considerando la lista de la figura 5, son:

x = a.buscarDatos('f', y) entonces x queda valiendo 9 y
y.retornarDatos() valdrá 8.
x = a.buscarDatos('u', y) entonces x queda valiendo 5 y
y.retornarDatos() valdrá 3.
x = a.buscarDatos('b', y) entonces x queda valiendo 4 y
y.retornarDatos() valdrá None.
x = a.buscarDatos('m', y) entonces x queda valiendo None y
y.retornarDatos() valdrá ultimo.

El método **borrar(x, y)** controla que el parámetro **x** sea diferente de **None**: si **x** es **None**, produce el mensaje de que el dato **d** (el dato buscado con el método buscarDatos) no se halla en la lista y retorna; si **x** es diferente de **None**, invoca el método **desconectar(x, y)**.

El método **desconectar(x, y)** simplemente desconecta el nodo **x** de la lista que invoca el método. Para desconectar un nodo de una lista se necesita conocer cuál es el nodo anterior. Por tanto, en el nodo **y**, el segundo parámetro, su campo de dato contiene el nodo anterior a **x**. Si ejecutamos las siguientes instrucciones sobre la lista de la figura 5:

```
d = 'i'  
x = a.buscarDatos(d, y) // x queda valiendo 7 y  
y.retornarDatos() valdrá 9  
a.borrar(x, y) // desconecta el nodo x
```

Esta queda así:

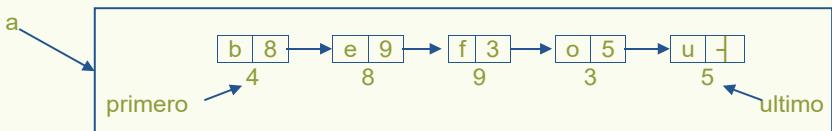


Figura 6

Un algoritmo de ejemplo de uso de la clase **LSL** es el siguiente:

```
1. a = LSL()
2. for i in range(1, 10):
3.     d = input("Entre dato: ")
4.     y = a.buscarDondeInsertar(d)
5.     a.insertar(d, y)
6.     d = input("Entre más datos: ")
7.     while d != "0":
8.         a.agregarDato(d)
9.         d = input("Entre más datos: ")
10.    a.recorrerLista()
11.    l = a.longitud()
12.    print(l)
13.    y = nodoSimple()
14.    x = a.buscarDato("a", y)
15.    a.borrar(x, y)
16.    print("despues de borrar primer vez")
17.    a.recorrerLista()
18.    l = a.longitud()
19.    print(l)
20.    x = a.primerNodo()
21.    a.borrar(x)
22.    print("despues de borrar segunda vez")
23.    a.recorrerLista()
24.    l = a.longitud()
25.    print(l)
26.    x = a.buscarDato("z", y)
27.    a.borrar(x)
28.    print("despues de borrar tercera vez")
29.    a.recorrerLista()
30.    l = a.longitud()
31.    print(l)
```

```
x = t.offset();
x = o.left;
y = o.top;

ax = settings.accX;
ay = settings.accY;
th = t.height();
wh = w.height();
tw = t.width();
ww = w.width();

(y + th + ay >= b &&
y <= b + wh + ay &&
x + tw + ax >= a &&
x <= a + ww + ax) {

    //trigger the collision
    if (!t.appeared) {
        //it scrolled out of view
        t.appeared = false;
    }
}

//create a modified function
modifiedFn = function() {
    //mark the element as appeared
    t.appeared = true;
}

//is this supposed to happen?
if (settings.one) {
    ...
    ...
    ...
    //remove the character
}
```

En la instrucción 1 se define un objeto de la clase **LSL**.

En las instrucciones 2 a 5 se construye la lista de tal forma que los datos entrados queden ordenados ascendenteamente a medida que se efectúa la construcción.

En las instrucciones 6 a 9 se entran datos agregándolos siempre al final de la lista.

Con la instrucción 10 se muestra la lista construida. Observe que los primeros nueve datos estarán ordenados ascendenteamente, mientras que los siguientes estarán en el orden en que fueron entrados.

Dejo al estudiante la tarea de analizar y racionalizar las instrucciones 11 a 31.

Se recomienda entender bien el algoritmo del método **borrar**.

Y, por último, dese cuenta de que usted puede hacer algoritmos con listas con solo saber qué hacen los métodos definidos para la clase.