uc3m | Universidad Carlos III de Madrid

OpenCourseWare
Grado Ingeniería Informática
Estructura de Datos y Algoritmos

Tema 2 Tipos Abstractos de Datos Lineales: 2.2. Listas Enlazadas

Profesores EDA

Objetivos

- Conocer las ventajas y desventajas de las estructura lista de Python.
- Conocer el concepto de Nodo y Lista enlazada.
- Saber implementar el TAD Pila basado en lista enlazada.
- Saber implementar el TAD Cola basado en lista enlazada.
- Ser capaces de discutir las ventajas y desventajas de estas implementaciones comparadas con las implementaciones basadas en arrays.

Objetivos

- Estudiar y conocer el TAD Lista
- Implementar el TAD Lista con una lista simplemente enlazada.
- Conocer el concepto de nodo doble y lista doblemente enlazada.
- Implementar el TAD Lista con una lista doblemente enlazada.
- Ser capaz de discutir sobre las ventajas y desventajas de cada una de las implementaciones estudiadas durante el tema.

Índice

- Lista de Python (array)
- Lista Enlazada
  - Nodo Simple
  - Lista Simplemente Enlazada
- Implementación del TAD Pila basada en lista enlazada.
- Implementación del TAD Cola basada en lista enlazada.
- Definición TAD Lista.
- Implementación TAD Lista usando lista simplemente enlazada.
- Implementación TAD Lista usando lista doblemente enlazada.

Lista de Python (array)

- En otros lenguajes de programación, se llaman **arrays**.
- En un array, todos sus elementos ocupan posiciones contiguas en la memoria principal.
- Para representar una pila o una cola de tamaño n, necesitamos una lista de tamaño n.

Ventajas de los arrays

- Cada elemento puede ser identificado por la posición que ocupa en la estructura:

  A[0]=María
  A[1]=Pepa
  A[2]=Juan
  ....
  A[6]=Daniel

- Fácil y rápido acceso a todos sus elementos: A[i]

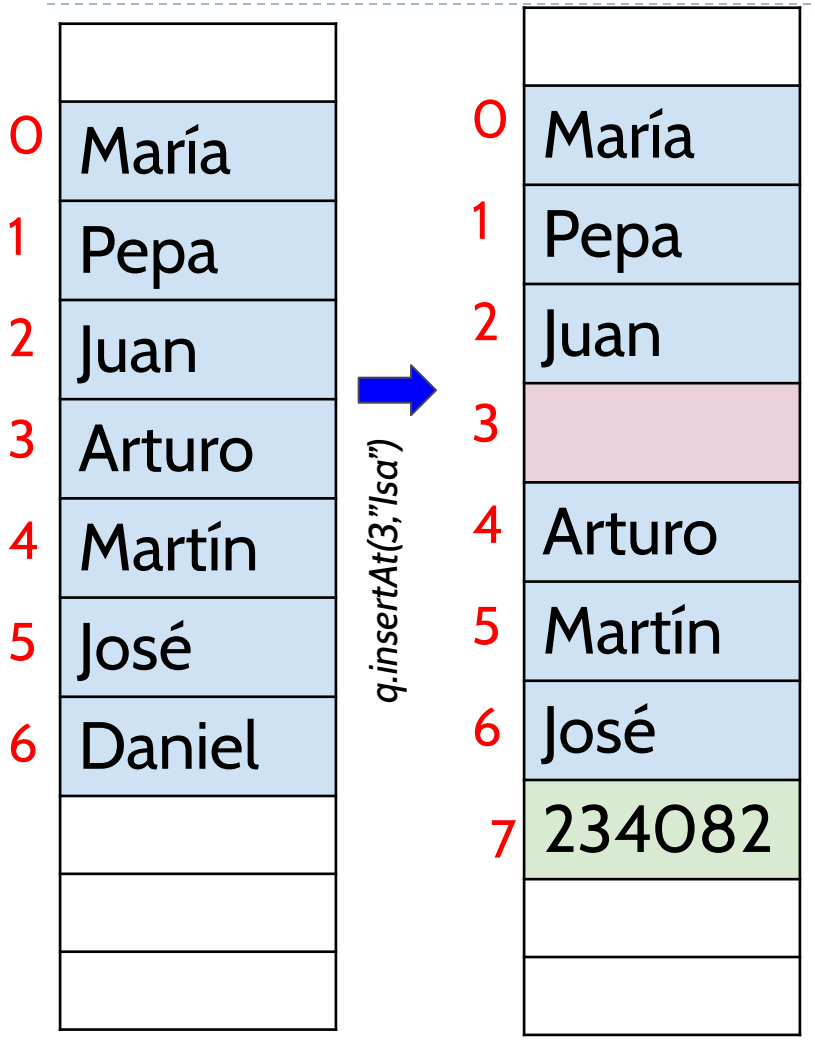
Desventajas de los arrays

- Si borramos el primer elemento de una cola (primer elemento del array), el intérprete de Python deberá mover todos los elementos del array a una posición anterior en la memoria principal.
- Si la cola tiene muchos elementos o si vamos a realizar muchas operaciones de borrado, **no es eficiente.**

Desventajas de los arrays

- Al tratar de insertar un elemento en mitad de la lista (por ejemplo, *q.insertAt(3, "Isa")*), el intérprete de Python deberá mover todos los elementos con un índice >=3 una posición a la derecha en la memoria principal. Después en la posición 3, deberá almacenar el nuevo elemento "Isa".
- No es eficiente si nuestro programa realiza muchas operaciones de inserción y/o el tamaño de la estructura lineal es muy grande.

Desventajas de los arrays



- Además, podría darse el caso que la posición contigua al último elemento de la lista, ya estuviera ocupada por otro proceso.
- En nuestro ejemplo, no se podría copiar el elemento "Daniel" a la siguiente posición en memoria.
- En ese caso, el administrador de memoria de nuestro sistema operativo debería buscar otra localización para almacenar todos los elementos del array de forma contigua.
- No es eficiente, si la lista es muy grande o se realizan muchas inserciones.

Desventajas de los arrays

- La misma problemática vamos a tener cuando intentamos apilar en una pila, encolar en una cola o añadir al final en una lista de Python.
- Si la siguiente posición de memoria ya está siendo utilizada por otro proceso, el administrador de memoria de nuestro sistema operativo deberá localizar otro espacio en memoria lo suficientemente grande como para almacenar todos los elementos de la lista.
- Será necesario mover todos los elementos a la nueva zona de memoria.
- **No es eficiente**, si la estructura lineal tiene muchos elementos o realizamos muchas veces este tipo de operaciones.

En resumen

- Los arrays (listas de Python) permiten un fácil y rápido acceso a sus elementos, porque estos se almacenan en posiciones contiguas en la memoria principal.
- Sin embargo, este almacenamiento contiguo en memoria principal, también provoca que algunas operaciones sean poco eficientes, porque el administrador de memoria de nuestro sistema operativo de Python va a necesitar mover los elementos de la estructura a nuevas posiciones de la memoria donde sea posible albergar todos los elementos de forma contigua.

Pregunta

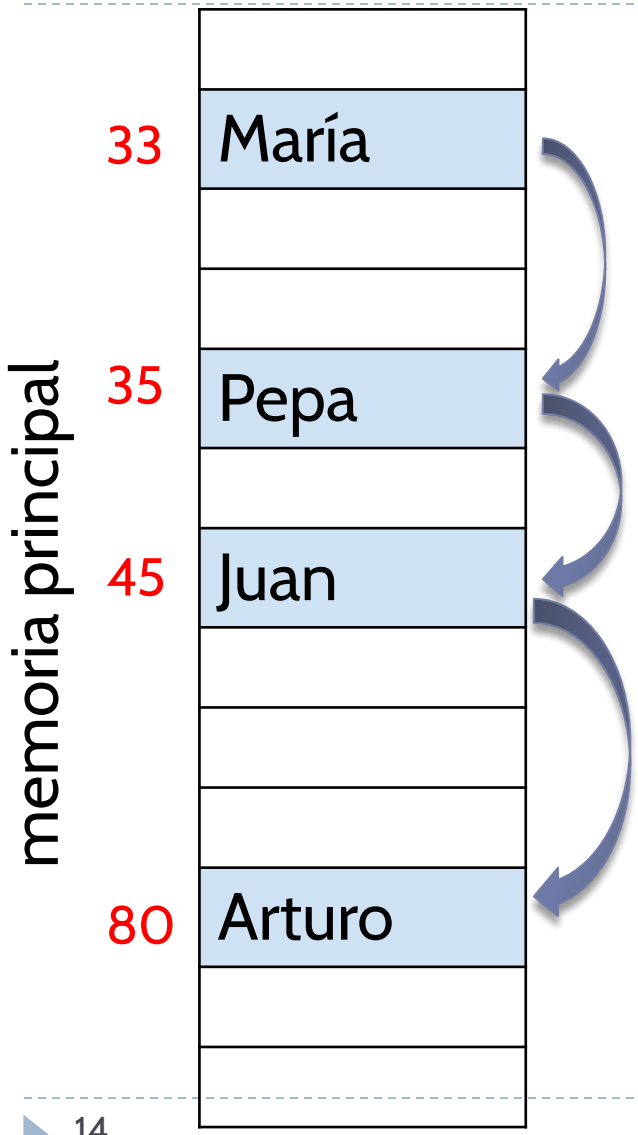
- Ya hemos visto que el almacenamiento en posiciones contiguas de memoria tiene sus ventajas (rápido acceso) y desventajas (poco eficiente en algunos casos).

- ¿Podríamos almacenar los elementos de una estructura lineal de forma no contigua en la memoria principal?.

Índice

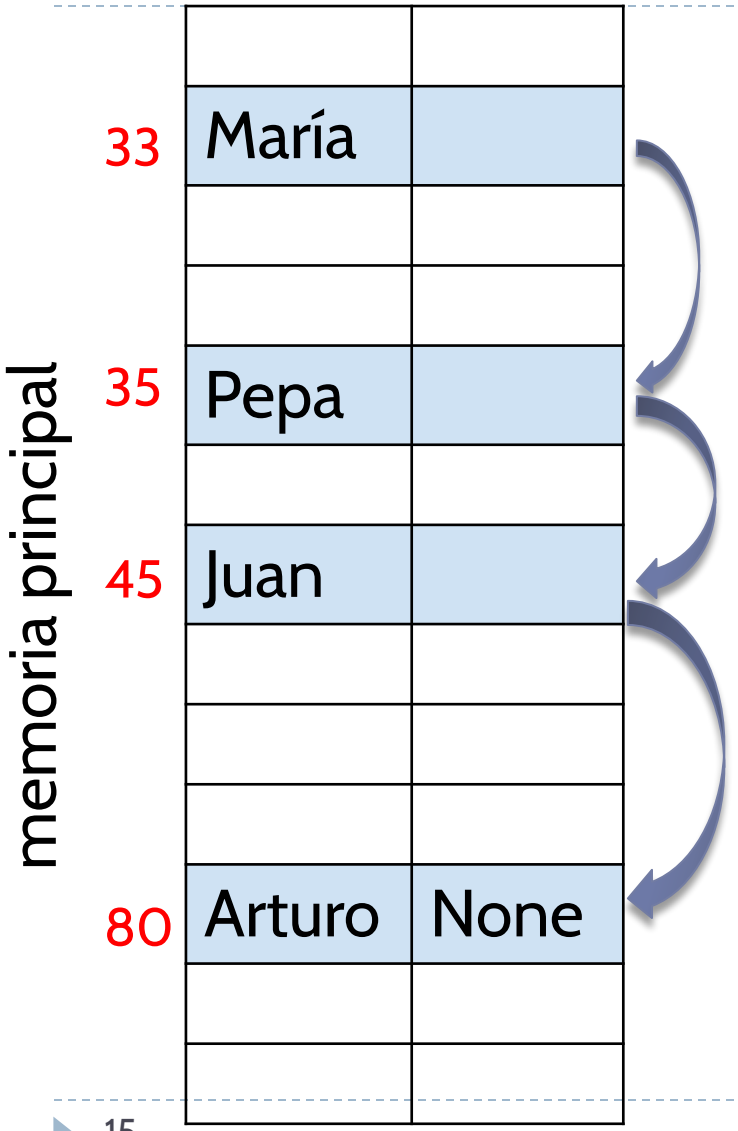
- Lista de Python (array)
- Lista Enlazada
  - Nodo Simple
  - Lista Simplemente Enlazada
- Implementación del TAD Pila basada en lista enlazada.
- Implementación del TAD Cola basada en lista enlazada.
- Definición TAD Lista.
- Implementación TAD Lista usando lista simplemente enlazada.
- Implementación TAD Lista usando lista simplemente enlazada.

Lista Enlazada



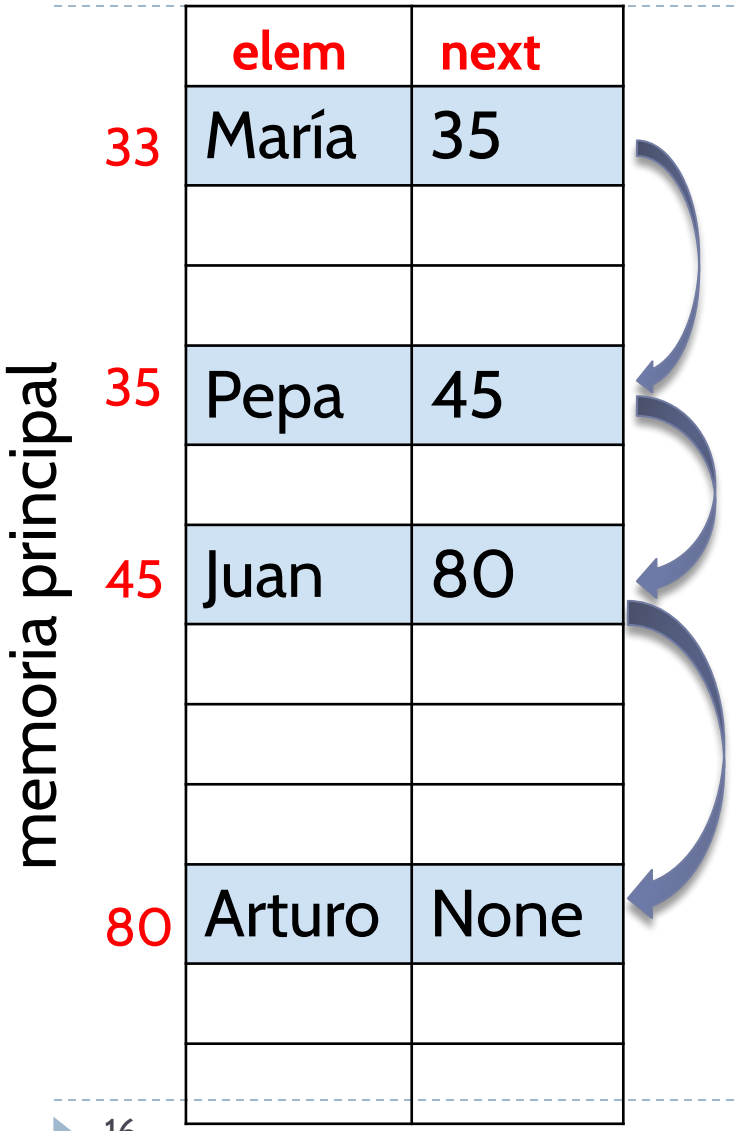
- Vamos a permitir que los elementos de nuestra estructura lineal sean almacenados en posiciones no contiguas de la memoria principal.
- Es decir, vamos a permitir saltos entre los distintos elementos.
- ¿Cómo almacenar la información de los saltos?

Lista Enlazada



- Además de almacenar cada elemento de la estructura lineal, también almacenaremos la posición del siguiente elemento en dicha estructura lineal.

Lista Enlazada



- Necesitamos implementar (en Python) una nueva estructura que nos permita almacenar un elemento y la referencia dónde se encuentra el siguiente elemento.

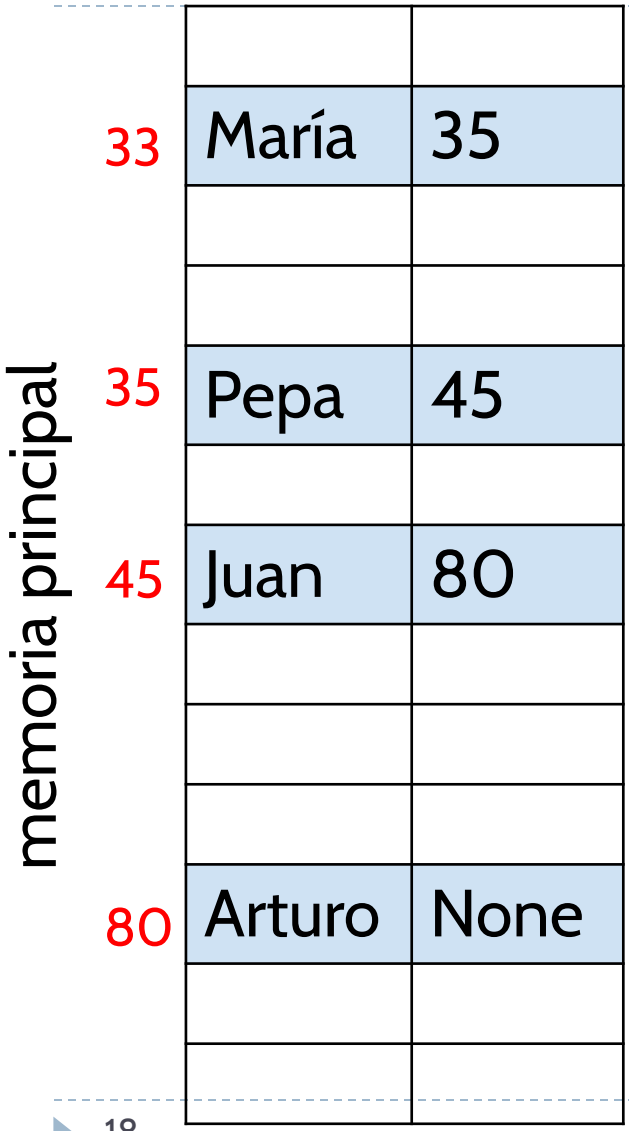
elem	next
María	35

Clase Python para Nodo simple

elem	next
María	35

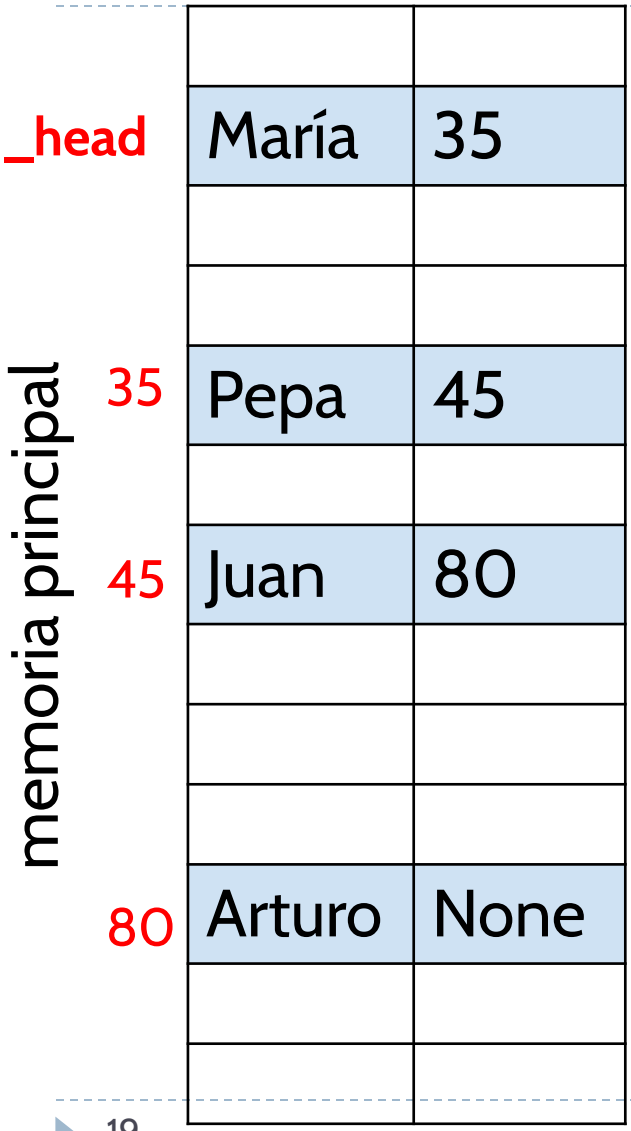
```
class SNode:
    def __init__(self, e: object, next_node: 'SNode' = None) -> None:
        self.elem = e
        self.next = next_node
```

Lista Enlazada



- Y para implementar una lista enlazada, ¿qué necesitamos?
- Necesitamos una clase de Python, que podemos llamar SList,
- ¿Y qué atributos necesitas definir en esa clase?

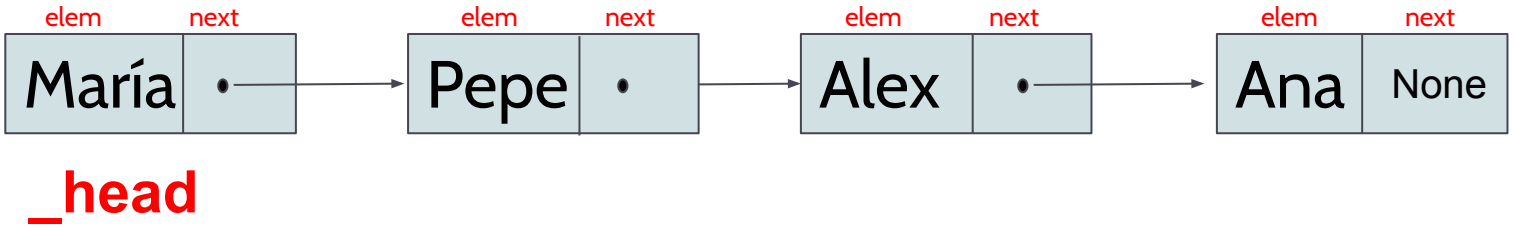
Lista Enlazada



- Sería suficiente con guardar la referencia, **_head**, al **primer nodo de la lista enlazada**.
- Más adelante veremos qué otros atributos podemos definir y qué ventajas nos ofrecen.

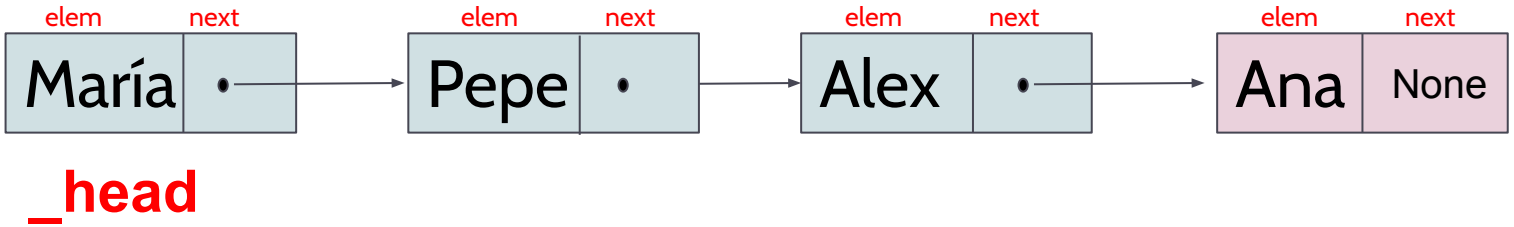
Lista Simplemente Enlazada

- Una lista enlazada está formada por una secuencia de nodos (simples) enlazados.
- Cada nodo almacena un elemento y la referencia al siguiente nodo.
- **_head** es la referencia al primer nodo de la lista, que nos permitirá acceder al resto.



Lista Simplemente Enlazada

- Sabremos que estamos en el último nodo de la lista porque su referencia next será None.



- Si la lista está vacía, _head será None.

Índice

- Lista de Python (array)
- Lista Enlazada
  - Nodo Simple
  - Lista Simplemente Enlazada
- Implementación del TAD Pila basada en lista enlazada.
- Implementación del TAD Cola basada en lista enlazada.
- Definición TAD Lista.
- Implementación TAD Lista usando lista simplemente enlazada.
- Implementación TAD Lista usando lista simplemente enlazada.

Implementación TAD Pila basada en lista enlazada

- La idea es utilizar una lista enlazada para almacenar los elementos de la pila, en lugar de una lista de Python.
- Es decir, cada elemento de la pila será almacenado en un nodo simple.
- ¿Dónde almacenamos la cima de la pila?: en el primer nodo o en el último nodo.

Implementación TAD Pila basada en lista enlazada

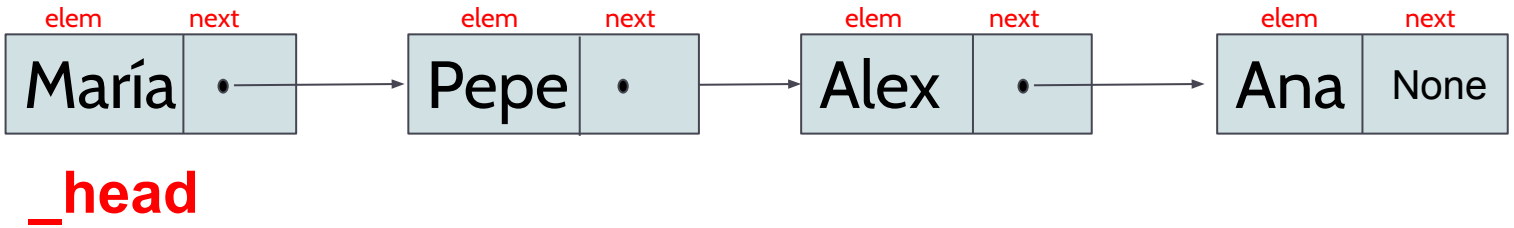
- La mejor opción es almacenar la cima de la pila en el primer nodo de la lista enlazada, es decir, en el nodo _head.
- Permitirá que las operaciones push y pop se hagan de forma más eficiente.
- Si por el contrario, decidimos almacenar la cima de la pila en el último nodo de la lista, siempre sería necesario recorrer toda la lista enlazada para poder realizar las operaciones push o pop.

Implementación TAD Pila basada en lista enlazada

- Como hemos mencionado antes nuestra pila será implementará como una lista enlazada.
- Tendrá un único atributo _head, y en el constructor será inicializado a None.

Implementación TAD Pila basada en lista enlazada

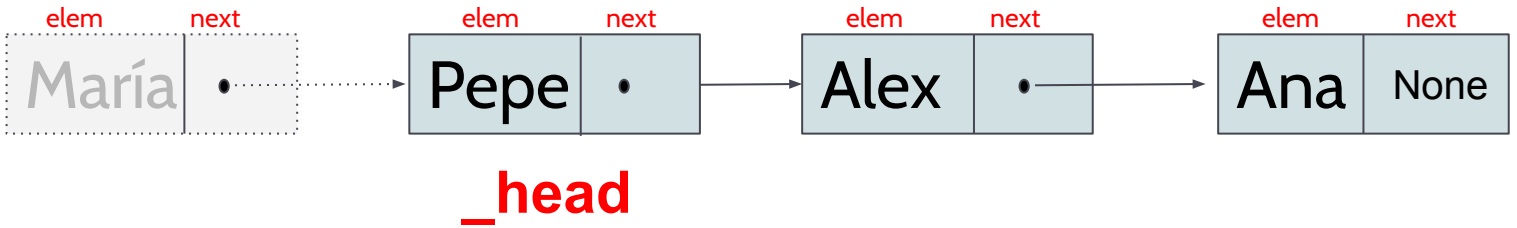
- El método **pop**, debe devolver y eliminar el elemento que hay en la cima de la pila. Para ello, debemos acceder al elemento que hay almacenado en el nodo _head y lo guardará en una variable que luego será retornada por el método.




```
result = _head.elem   # María
```

Implementación TAD Pila basada en lista enlazada

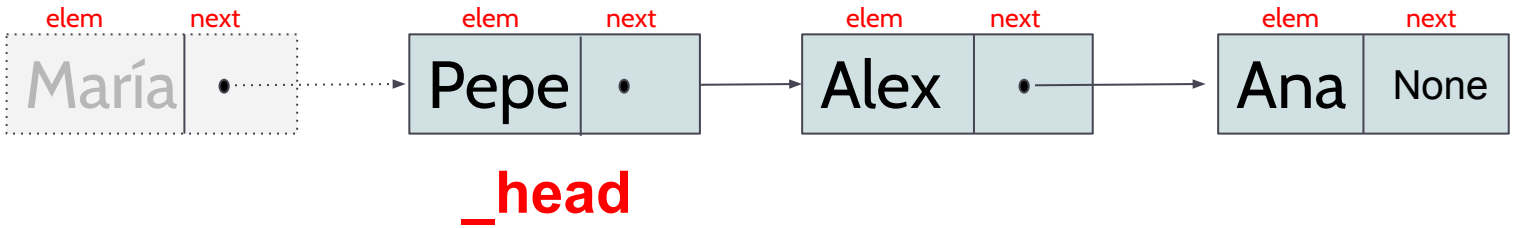
- Para que el método **pop** elimine la antigua cima de la pila, bastará con mover _head a su siguiente nodo.




```
_head = _head.next
```

Implementación TAD Pila basada en lista enlazada

- ¿Qué hacemos con el antiguo _head?. En realidad, no tenemos que preocuparnos, porque el recolector de basura de Python se encargará de liberar este espacio de memoria que no está siendo referenciado.



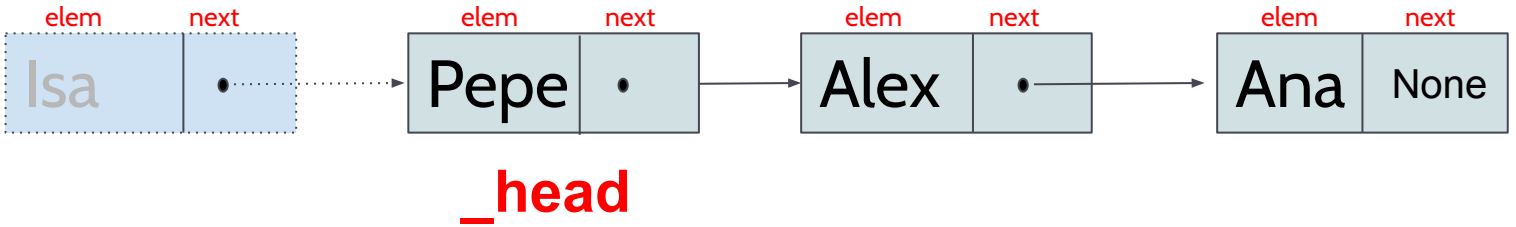

```
_head = _head.next
```

Recolector de basura

- El intérprete de Python dispone de un recolector de basura encargado de liberar el espacio de memoria que no está siendo utilizada por el programa en ejecución.
- El recolector de basura liberará el espacio ocupado por el que era el primer nodo (cuyo elemento es María), porque ya no está siendo referenciado por ninguna variable.
- Esta liberación de memoria es oculta para el programador. En otras palabras, no tienes que preocuparte ni de reservar ni de liberar memoria.
- Con otros lenguajes de programación (C, C++), la gestión de memoria sí es obligatoria.

Implementación TAD Pila basada en lista enlazada

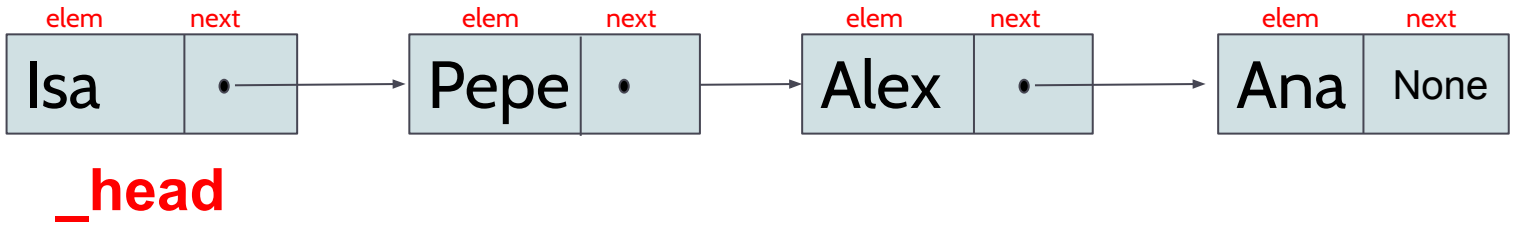
- El método **push**, recibirá un elemento (por ejemplo, 'Isa'). En primer lugar, creará un nuevo nodo que apuntará al nodo _head, donde se almacena la cima de la pila.




```
new_node = SNode("Isa", _head)
```

Implementación TAD Pila basada en lista enlazada

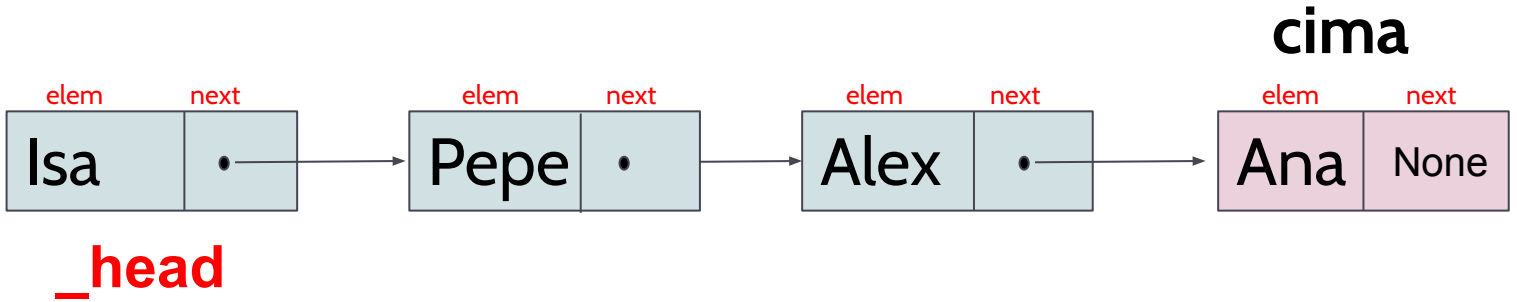
- El último paso en el método **push**, será actualizar _head para sea el nuevo nodo que acabamos de añadir al principio de la lista enlazada.




```
_head = new_node
```

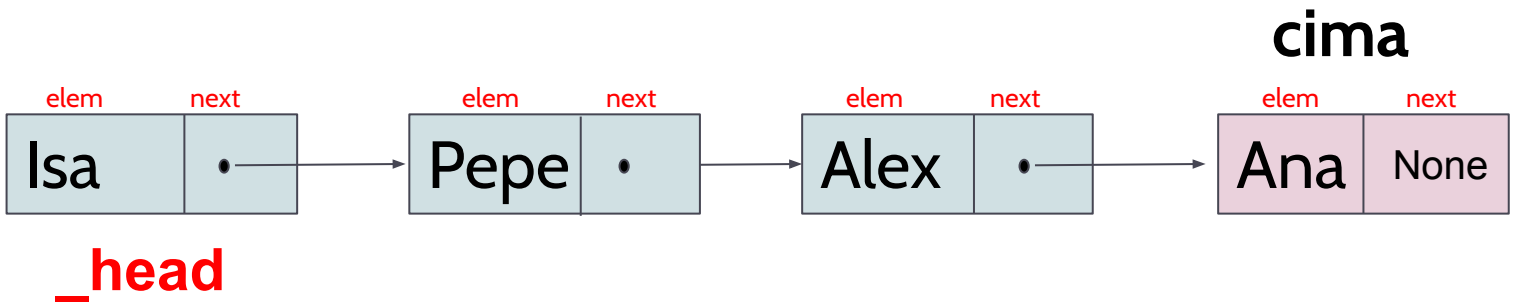
Implementación TAD Pila basada en lista enlazada

- Si la cima se almacenará en el último nodo de la lista, no sería eficiente, ¿por qué?.



Implementación TAD Pila basada en lista enlazada

- Si la cima está en el último nodo de la lista enlazada, no sería eficiente, porque tanto para apilar (push) como para desapilar (pop), sería necesario siempre recorrer toda la lista para poder añadir un nuevo nodo al final de la lista o borrar el último nodo.



Implementación TAD Pila basada en lista enlazada

Solución

- Observa que en la implementación proporcionada también se utiliza un segundo atributo, _size, que nos permite almacenar el número de elementos en la pila (o número de nodos en la lista enlazada).
- Date cuenta que cada vez que apilas o desapilas, en la implementación de estas operaciones se debe actualizar el valor del atributo _size, según corresponda.

Implementación TAD Pila basada en lista enlazada

Solución

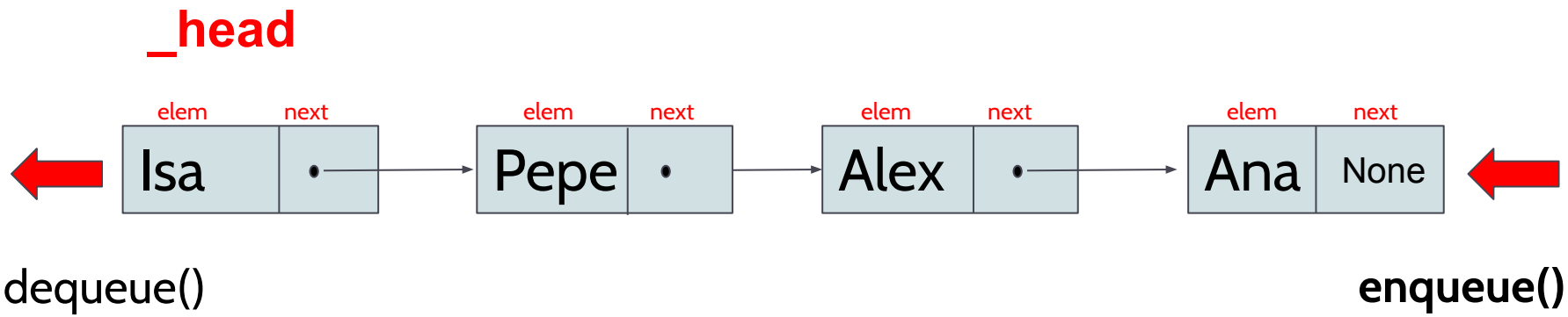
- La principal ventaja que nos ofrece el atributo _size es que nos permite conocer el tamaño de la pila de forma inmediata. Si no tuviéramos este atributo, cada vez que necesitamos conocer el tamaño de la pila, deberíamos recorrer toda la lista enlazada y contar sus nodos.

Índice

- Lista de Python (array)
- Lista Enlazada
  - Nodo Simple
  - Lista Simplemente Enlazada
- Implementación del TAD Pila basada en lista enlazada.
- Implementación del TAD Cola basada en lista enlazada.
- Definición TAD Lista.
- Implementación TAD Lista usando lista simplemente enlazada.
- Implementación TAD Lista usando lista simplemente enlazada.

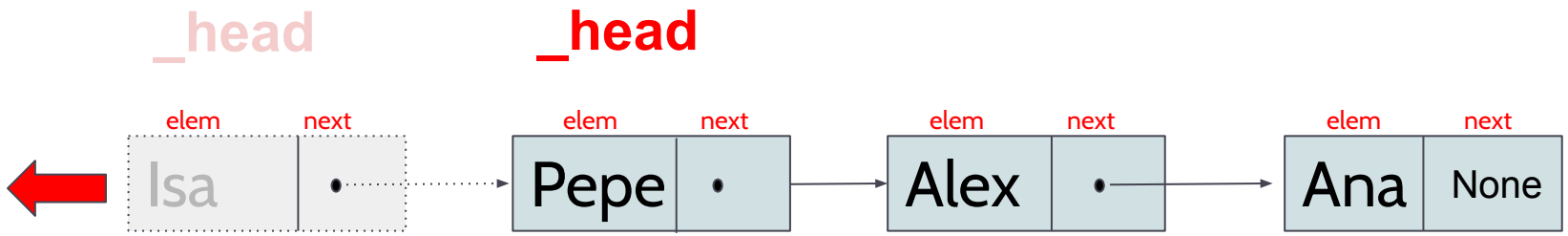
Implementación TAD Cola basada en lista enlazada

- La idea es que el nodo _head, primer nodo de la lista, contendrá el primer elemento que entró en la cola, y este elemento será el próximo en salir.
- El último elemento de la cola se almacenará en el último nodo de la lista enlazada.



Implementación TAD Cola basada en lista enlazada

- El método **dequeue**() devolverá el valor del primer nodo ('Isa') y eliminará dicho nodo, moviendo la referencia _head al siguiente nodo (con elem Pepe). Su implementación es similar al método pop() de la clase Stack.

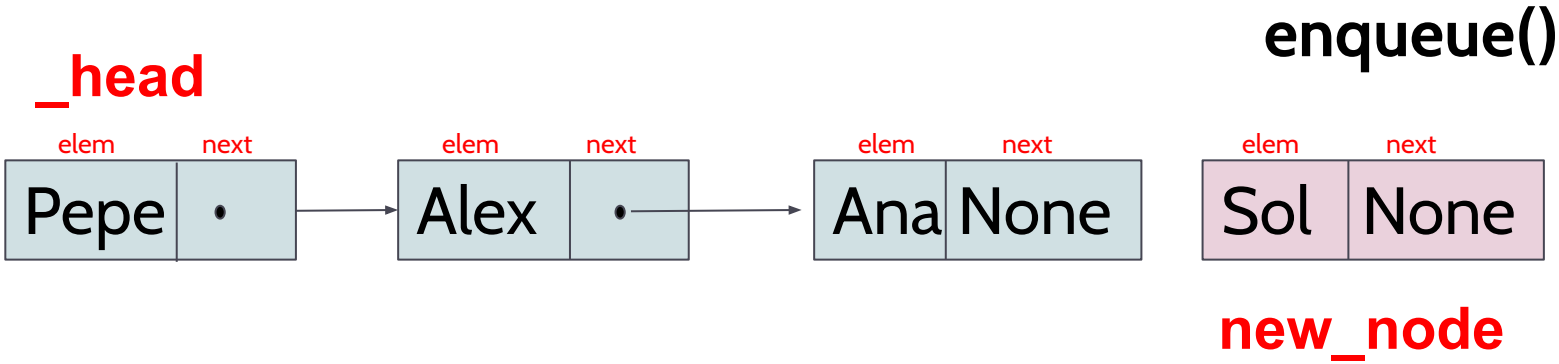


dequeue()

```
result = _head.elem
_head = _head.next
return result
```

Implementación TAD Cola basada en lista enlazada

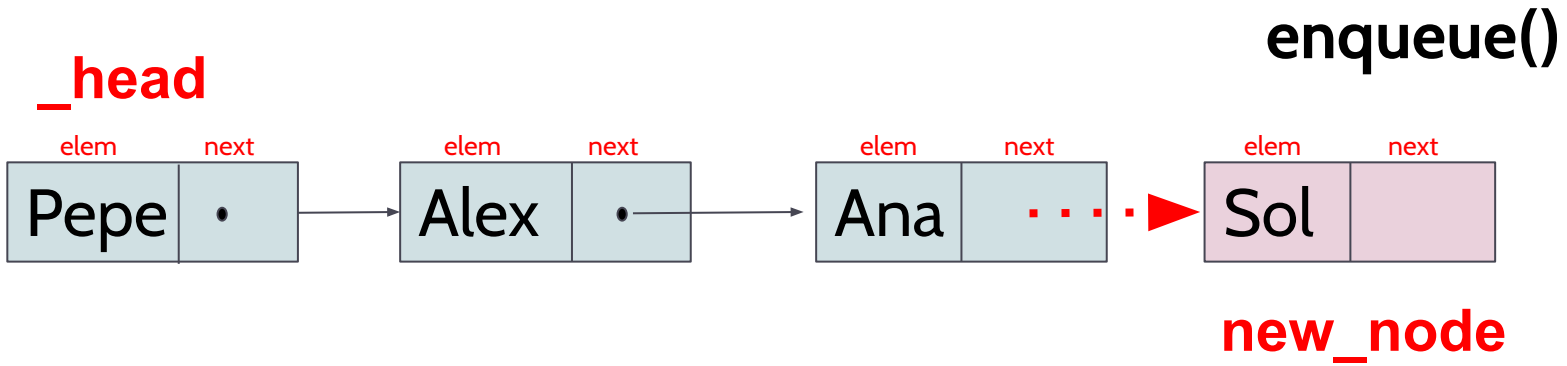
- El método **enqueue**(e) debe crear un nuevo nodo para que contenga el elemento e (por ejemplo, 'Sol') y añadirlo al final de la cola (después del nodo que contiene a 'Ana').




```
new_node = SNode('Sol', None)
```

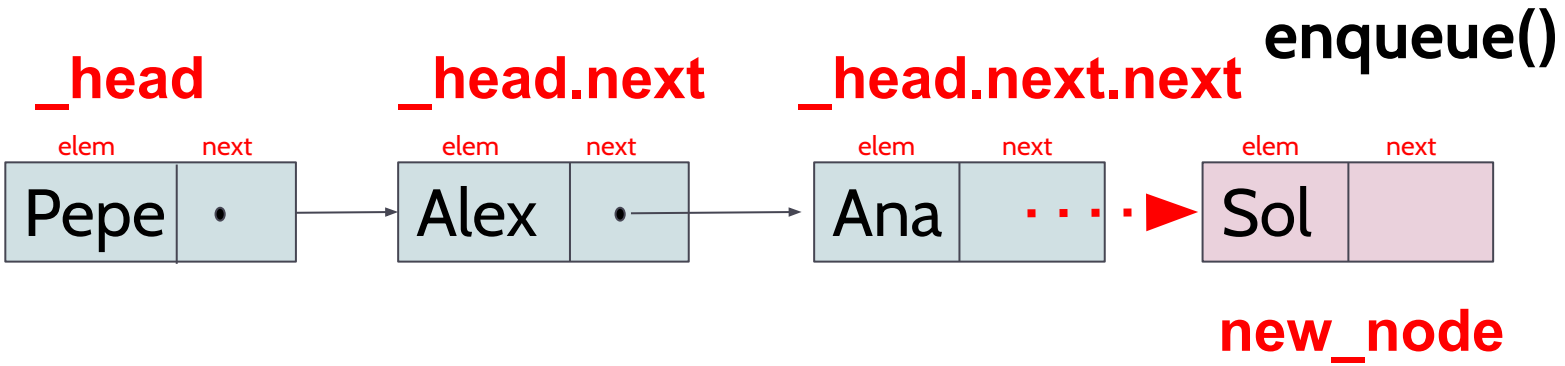
Implementación TAD Cola basada en lista enlazada

- ¿Cómo podemos enlazar el último nodo de la lista con el nuevo nodo?



Implementación TAD Cola basada en lista enlazada

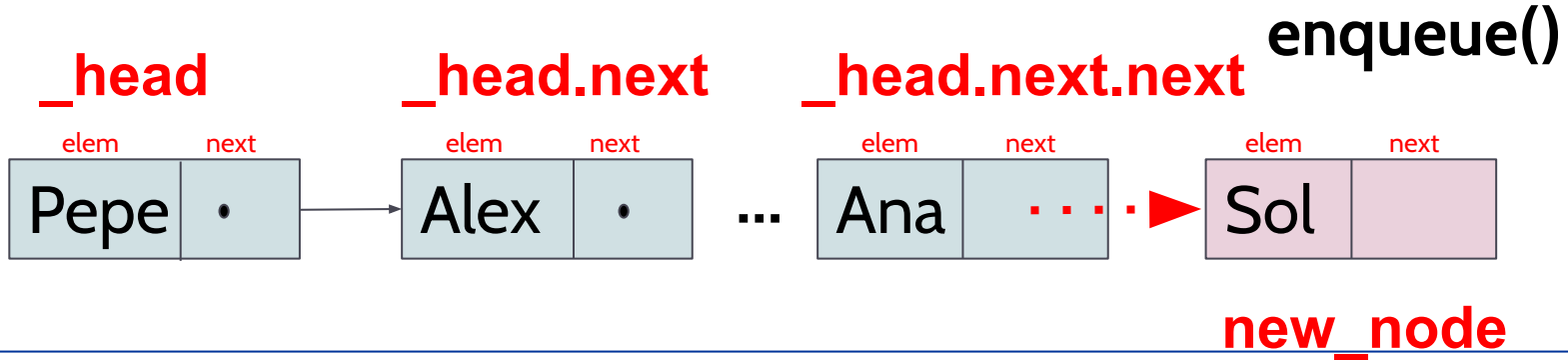
- ¿Cómo podemos enlazar el último nodo de la lista con el nuevo nodo?




```
last_node = _head.next.next
last_node.next = new_node
```

Implementación TAD Cola basada en lista enlazada

- ¿Qué pasa si en lugar de tener 3 nodos, tenemos 5 nodos en la lista?

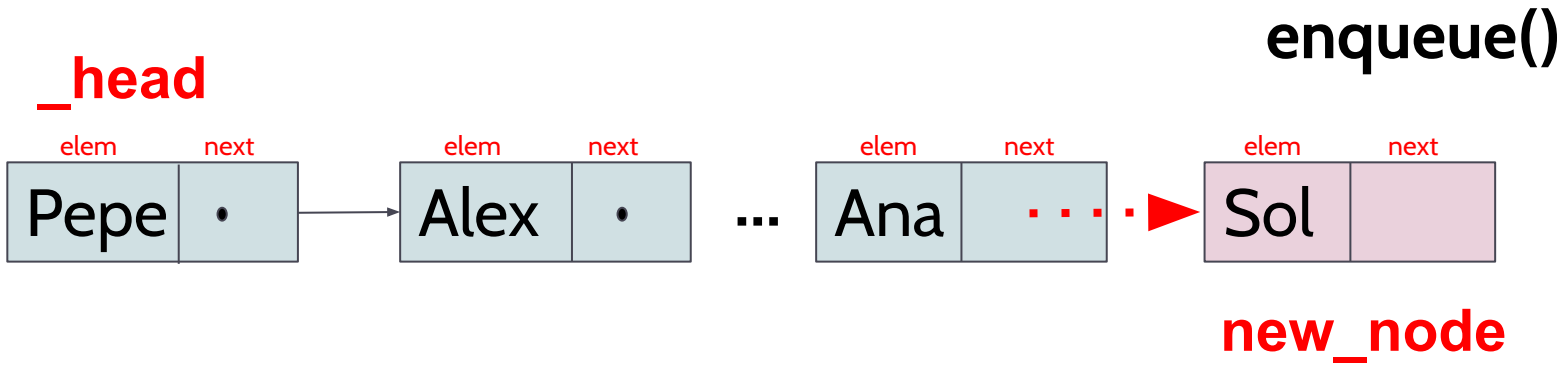



```
last_node = _head.next.next.next.next.next
last_node.next = new_node
```

- Código mal refactorizado!!!. El código debe ser general para que pueda resolver el problema independientemente del número de nodos!!!

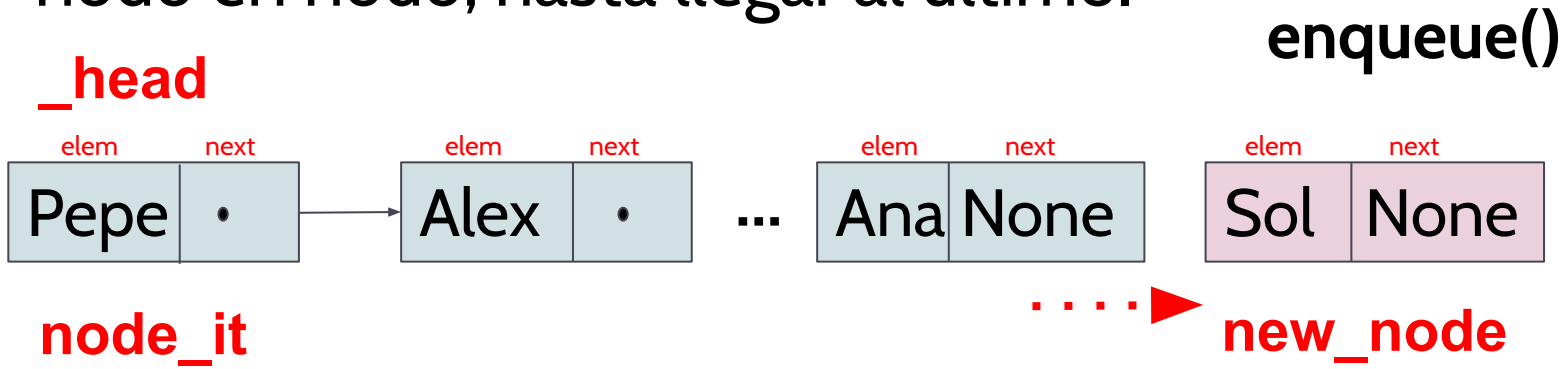
Implementación TAD Cola basada en lista enlazada

- ¿Cómo podemos alcanzar el último nodo?



Implementación TAD Cola basada en lista enlazada

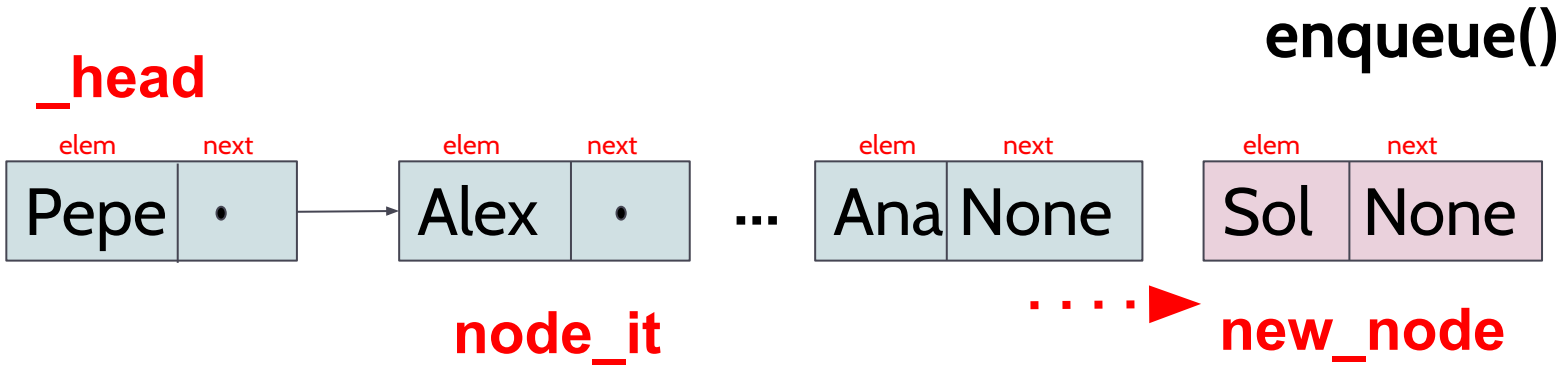
- Para recorrer una lista enladaza, puedes usar una variable auxiliar, **node_it**, cuyo valor inicial sea el nodo _head.
- El objetivo de esta variable auxiliar es ir saltando de nodo en nodo, hasta llegar al último.




```
node_it = _head
while … :
   node_it = node_it
```

Implementación TAD Cola basada en lista enlazada

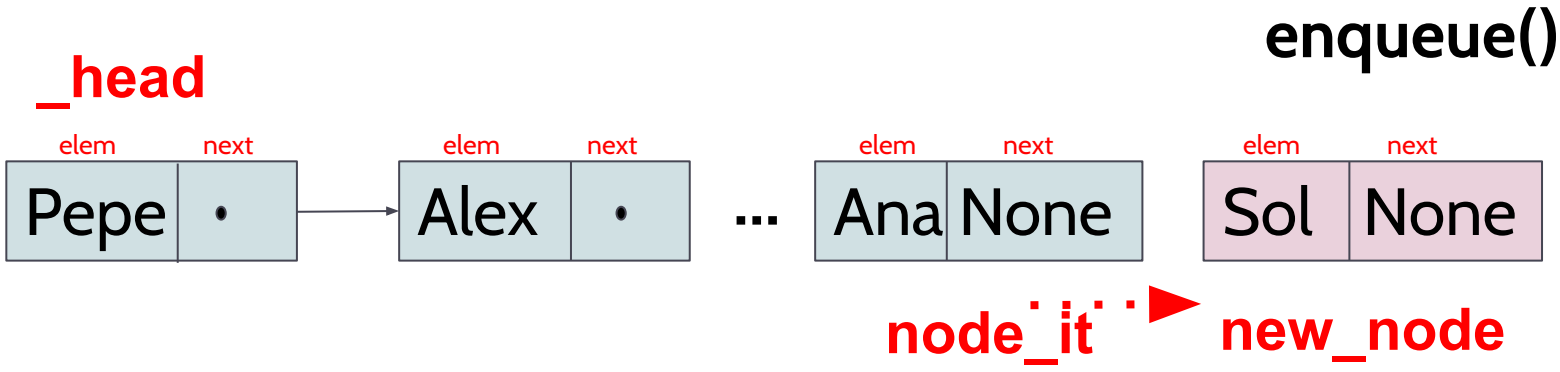
- El objetivo de esta variable auxiliar, node_it, es ir saltando de nodo en nodo, hasta llegar al último.




```
node_it = _head
while … :
   node_it = node_it
```

Implementación TAD Cola basada en lista enlazada

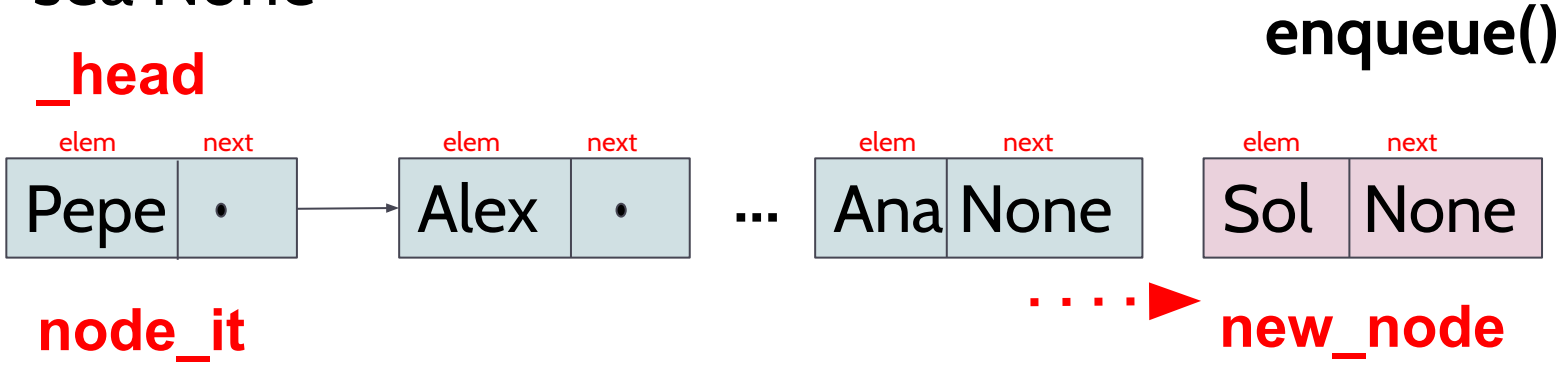
- El objetivo de esta variable auxiliar, node_it, es ir saltando de nodo en nodo, hasta llegar al último.




```
node_it = _head
while … :
   node_it = node_it
```

Implementación TAD Cola basada en lista enlazada

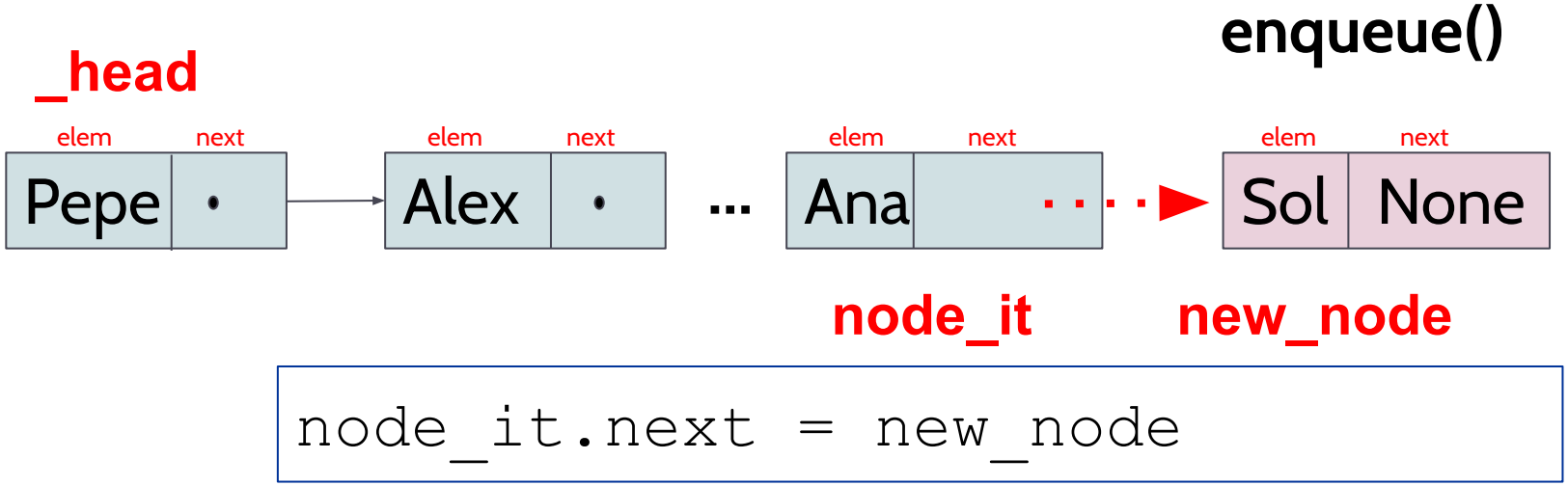
- ¿Cómo sabemos que **node_it** ya está en el último nodo de la lista?. Iteramos mientras node_it.next no sea None




```
node_it = _head
while node_it.next is not None:
   node_it = node_it
```

Implementación TAD Cola basada en lista enlazada

- Cuando node_it ya es el último nodo, simplemente tenemos que enlazar node_it con el nuevo nodo.

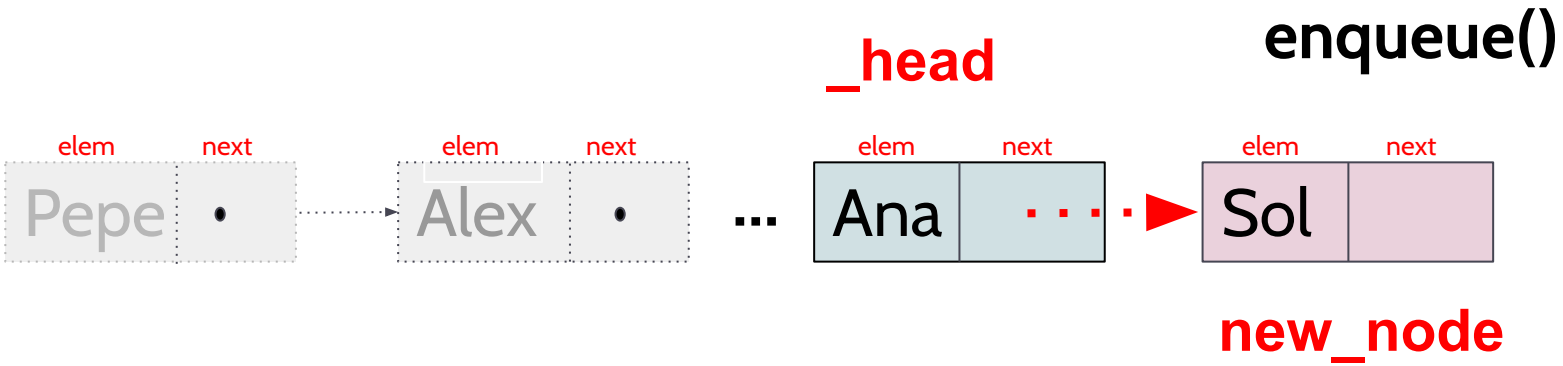



```
node_it.next = new_node
```

Implementación TAD Cola

Implementación TAD Cola basada en lista enlazada

- **Importante:** Para recorrer una lista enlazada, nunca uses el nodo _head.
- Si mueves el nodo _head, estarás modificando la estructura de la lista.



_head = _head.next

Análisis de la complejidad temporal

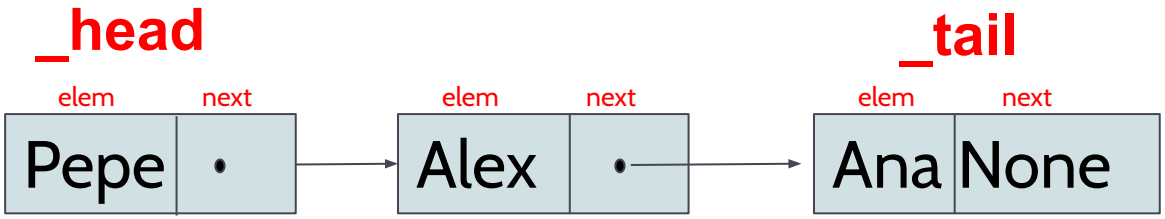
- **dequeue**() implica la ejecución de un número constante de instrucciones, y no depende del tamaño de la cola.
- Sin embargo, la función **enqueue**() tiene que recorrer toda la cola hasta alcanzar el último nodo. Su tiempo de ejecución va a depender del tamaño de la cola.

Análisis de la complejidad temporal

- ¿Cómo podríamos conseguir que **enqueue**() fuera más eficiente?
- Es decir, ¿cómo podemos añadir un nuevo elemento al final de la cola sin tener que recorrer toda la cola?

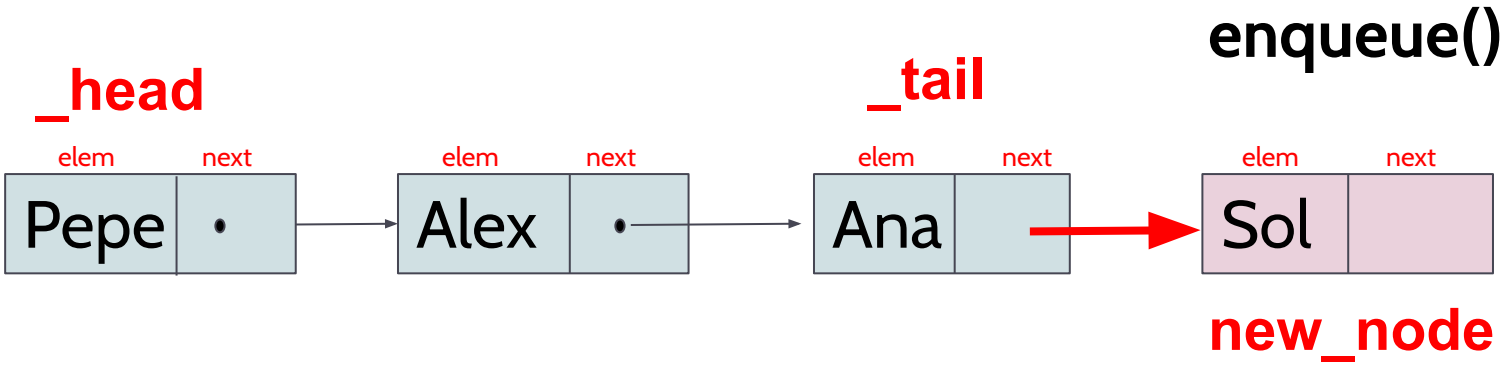
Una implementación más eficiente para TAD cola basada en lista enlazada

- La idea es utilizar un segundo atributo que siempre haga referencia al último nodo de la lista.



Una implementación más eficiente para TAD cola basada en lista enlazada

- De esta forma, solo necesitaremos enlazar el último nodo con el nuevo nodo a añadir.

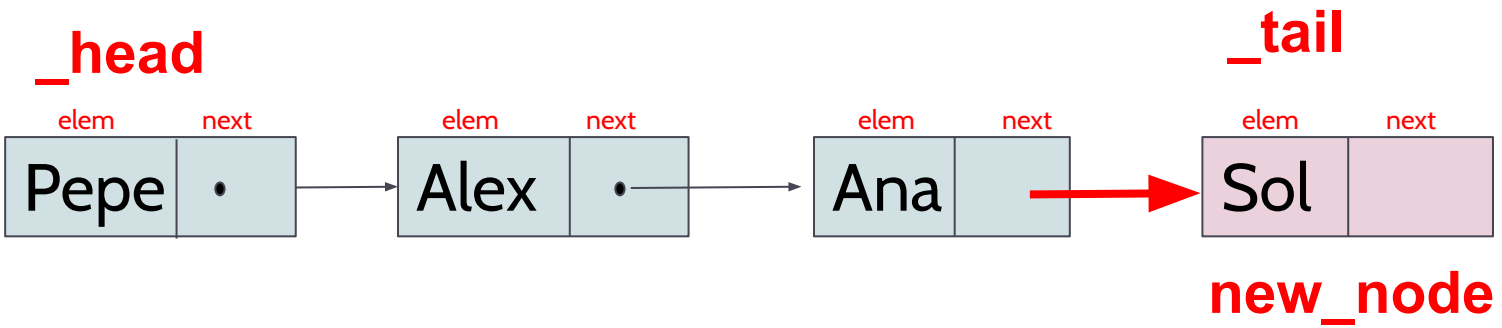



```
new_node = SNode('Sol')
_tail.next = new_node
```

Una implementación más eficiente para TAD cola basada en lista enlazada

- Por último, será necesario actualizar la referencia de _tail para que apunte al nuevo último nodo.

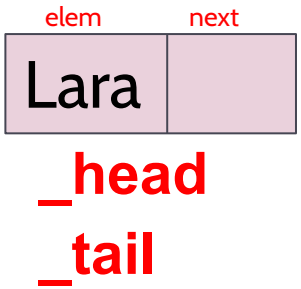
enqueue()




```
new_node = SNode('Sol')
_tail.next = new_node
tail = new_node
```

Una implementación más eficiente para TAD cola basada en lista enlazada (_head y _tail)

- Implementación de TAD Cola con una lista enlazada y con referencias _head y _tail
- En esta implementación, cuando se crea la cola, _head y _tail serán None.
- Una vez que se encola el primer elemento, _head y _tail deben referenciar al mismo nodo.



Refactoriza siempre que puedas

```
def enqueue(self, e: object) -> None:
    """Adds a new element, e, at the end of the queue"""
    new_node = SNode(e)
    if self.is_empty():
        self._head = new_node
        self._tail = new_node
        self._size += 1
    else:
        self._tail.next = new_node
        self._tail = new_node
        self._size += 1
```

Código repetido innecesariamente => Código más largo, más difícil de mantener y más propenso a errores durante el mantenimiento

- Refactoriza tu código!!! Refactorizar = crear un código más limpio
- **Nota**: _size no es obligatorio, pero consigue que el método _len_ sea más eficiente.

Refactoriza siempre que puedas

```
def enqueue(self, e: object) -> None:
    """Adds a new element, e, at the end of the queue"""
    new_node = SNode(e)
    if self.is_empty():
        self._head = new_node
    else:
        self._tail.next = new_node
    self._tail = new_node
    self._size += 1
```

- Código refactorizado, menos líneas, más fácil de entender y comprender.
- Aplicar Zen of Python

Análisis de la complejidad

- En el método enqueue(), ya no es necesario recorrer toda la cola.
- Ambos métodos, dequeue() y enqueue(), son eficientes porque sólo es necesario ejecutar un número constante de instrucciones, que no depende del tamaño de la cola.

Índice

- Lista de Python (array)
- Lista Enlazada
  - Nodo Simple
  - Lista Simplemente Enlazada
- Implementación del TAD Pila basada en lista enlazada.
- Implementación del TAD Cola basada en lista enlazada.
- **Definición TAD Lista.**
- Implementación TAD Lista usando lista simplemente enlazada.
- Implementación TAD Lista usando lista simplemente enlazada.

TAD Lista

- La idea del TAD Lista, es representar una secuencia de elementos, pero a diferencia de los TAD Cola o Pila, donde únicamente era posible operar en los extremos, en este nuevo TAD, las operaciones van a permitir operar en cualquier posición de la lista.

TAD Lista

- Definición informal:
  - Secuencia de elementos $\{a_1,a_2,...,a_n\}$ donde cada elemento tiene un único predecesor (excepto el primero que no tiene ninguno) y un único sucesor (excepto el último que no tiene ninguno).
  - Operaciones básicas:
    - **añadir** un elemento a la lista.
    - **borrar** un elemento de la lista.
    - **leer** un elemento de la lista.

TAD Lista (definición formal)

- Una colección de elementos donde cada elemento guarda una posición relativa con respecto al resto. Operaciones:
  - **List**() crea una lista vacía. En Python, __init__(self).
  - **add_first**(e) añade el elemento e al inicio de la lista.
  - **add_last**(e) añade el elemento e al final de la lista.

TAD Lista (definición formal)

- **remove_first**(): devuelve y elimina el primer elemento de la lista. Si la lista está vacía, muestra un mensaje de error y devuelve el objeto nulo (None).
- **remove_last**(): devuelve y elimina el último elemento de la lista. Si la lista está vacía, muestra un mensaje de error y devuelve el objeto nulo (None).
- **first**(): devuelve el primer elemento de la lista.
- **last**(): devuelve el último elemento de la lista.

TAD Lista (definición formal)

- **is_empty**(): devuelve cierto si la lista está vacía, Falso en otro caso.
- **len**(): devuelve el número de elementos de la lista.
- **index**(e): devuelve la primera posición del elemento e en la lista. Si el elemento no existe, devuelve -1.
- **getAt**(index): devuelve el elemento en la posición index de la lista. Si index está fuera de rango, muestra un mensaje de error y devuelve el objeto nulo (None).

TAD Lista (definición formal)

- **insertAt**(index, e): añade el elemento e en la posición index de la lista. Si index está fuera de rango, muestra un mensaje de error.
- **removeAt**(index): devuelve y borra el elemento de la posición index de la lista. Si index está fuera de rango, muestra un mensaje de error y devuelve el objeto nulo (None).

Índice

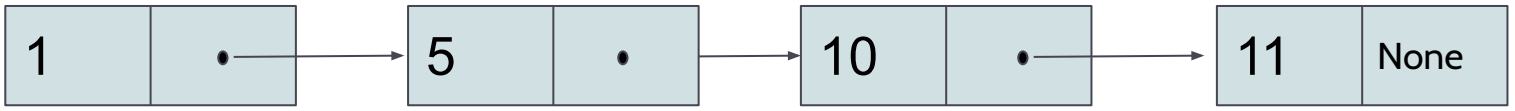
- Lista de Python (array)
- Lista Enlazada
  - Nodo Simple
  - Lista Simplemente Enlazada
- Implementación del TAD Pila basada en lista enlazada.
- Implementación del TAD Cola basada en lista enlazada.
- Definición TAD Lista.
- **Implementación TAD Lista usando lista simplemente enlazada.**
- Implementación TAD Lista usando lista simplemente enlazada.

Implementaciones del TAD Lista

- En realidad, ya conocemos una implementación del TAD Lista: las listas de Python (arrays).
- Hoy estudiaremos una nueva implementación para el TAD Lista basado en listas enlazadas.

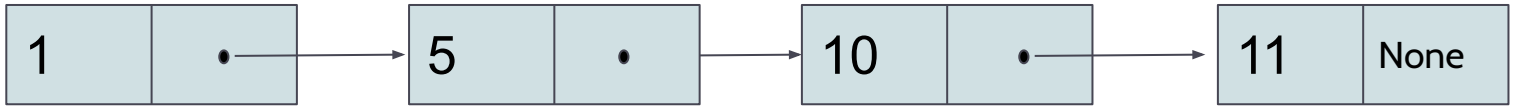
Posibles implementación TAD Lista

- lista simplemente enlazada con una única referencia al primer nodo, _head.



_head

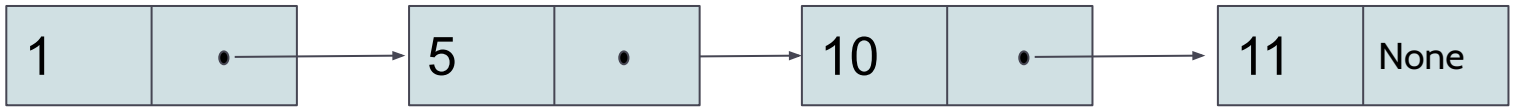
- lista simplemente enlazada con referencias al primer y último nodo: _head y _tail



_head **_tail**

Posibles implementación TAD Lista

- Elegimos la segunda implementación por la ventaja que supone a la hora de añadir un elemento al final de la lista



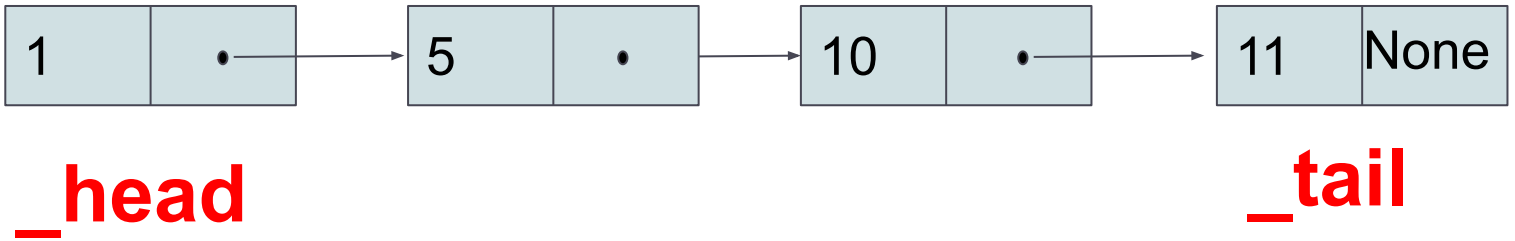
_head **_tail**

- También usaremos el atributo **_size**, porque consigue que el método **len** sea más eficiente (simplemente tiene que devolver el valor del atributo _size, sin necesidad de recorrer todos los nodos para calcular el tamaño de la lista).

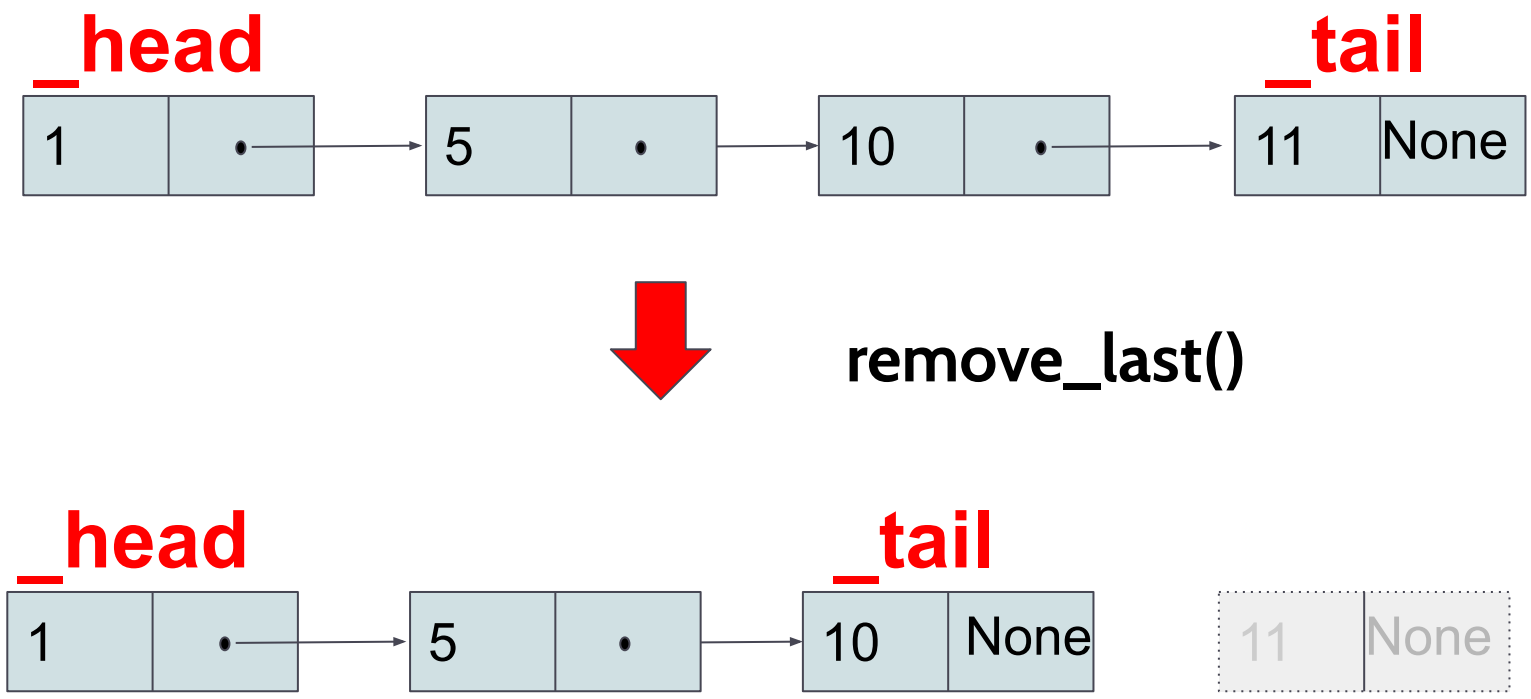
_size = 4

Implementación TAD Lista (_head y _tail)

- Las implementaciones de algunos métodos son similares a las vistas en las implementaciones de los TAD anteriores:
  - add_first(e) = push(e)
  - remove_rirst() = pop()/dequeue()
  - add_last(e) = enqueue(e)

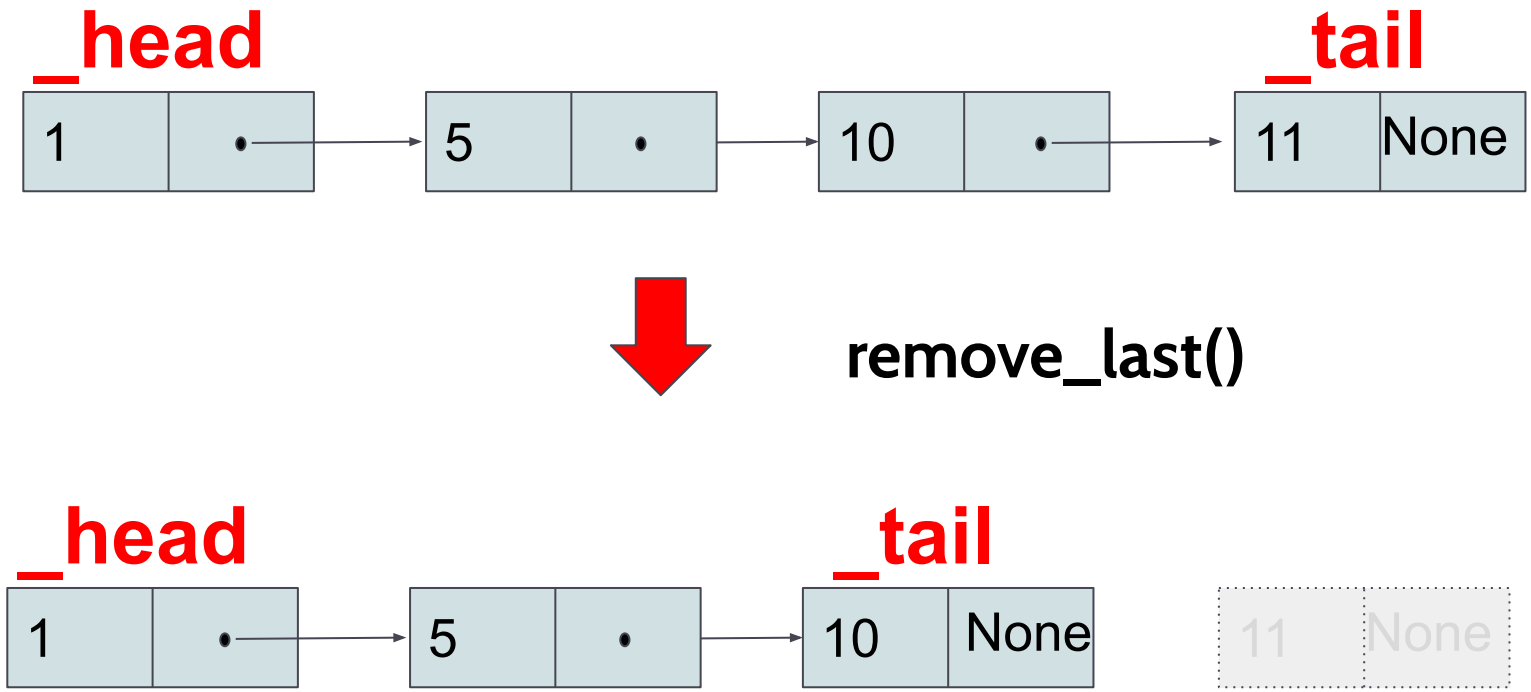


Implementación TAD Lista (_head y _tail)



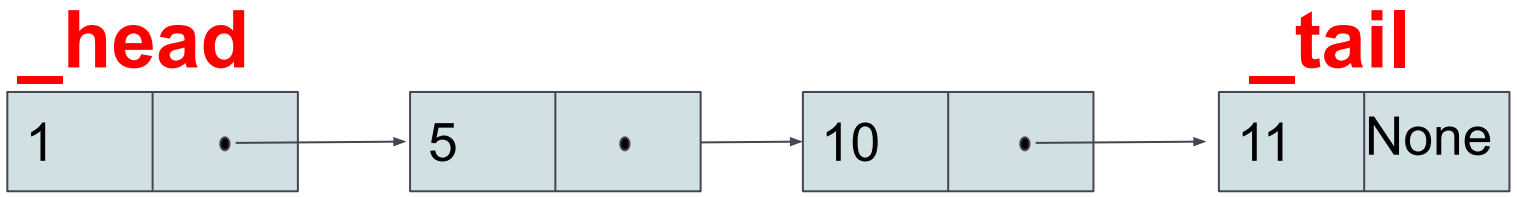
Implementación TAD Lista (_head y _tail)

- El penúltimo nodo será el nuevo último nodo. ¿Cómo podemos acceder al penúltimo nodo?



Implementación TAD Lista (_head y _tail)

- Es necesario recorrer la lista hasta alcanzar el penúltimo nodo.
- Definimos la variable prev_node para iterar sobre la lista

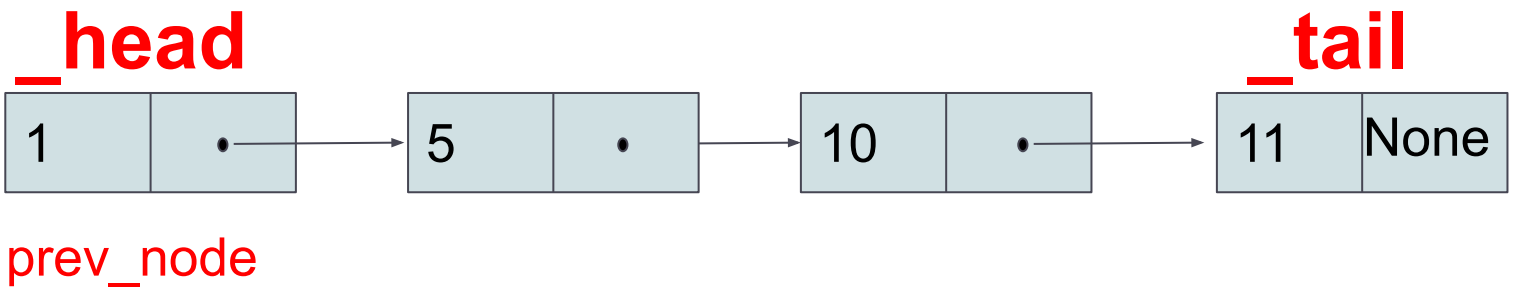


prev_node

```
result=self._tail.elem  # result=11
prev_node=self._head
…
return result
```

Implementación TAD Lista (_head y _tail)

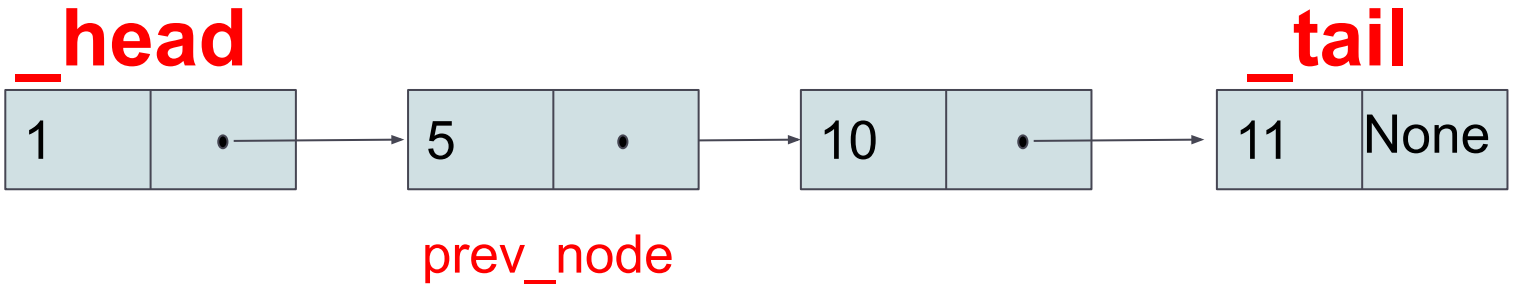
- Vamos iterando hasta alcanzar el penúltimo nodo




```
prev_node=self._head
while … :
    prev_node = prev_node.next
```

Implementación TAD Lista (_head y _tail)

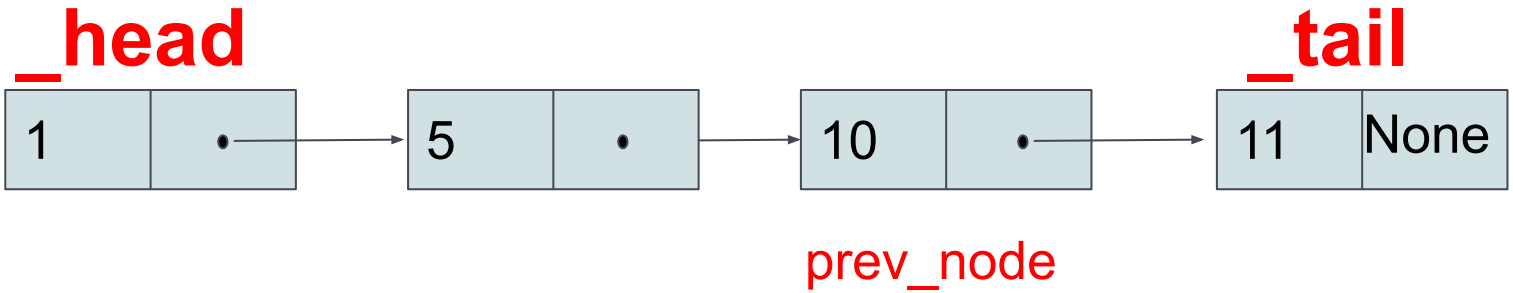
- Vamos iterando hasta alcanzar el penúltimo nodo, ¿Cuál debe ser la condición del bucle?




```
prev_node=self._head
while … :
    prev_node = prev_node.next
```

Implementación TAD Lista (_head y _tail)

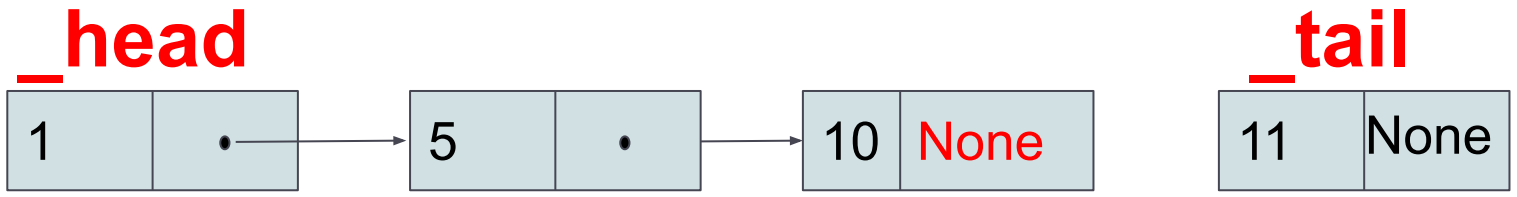
- Vamos iterando hasta alcanzar el penúltimo nodo, ¿Cuál debe ser la condición del bucle?



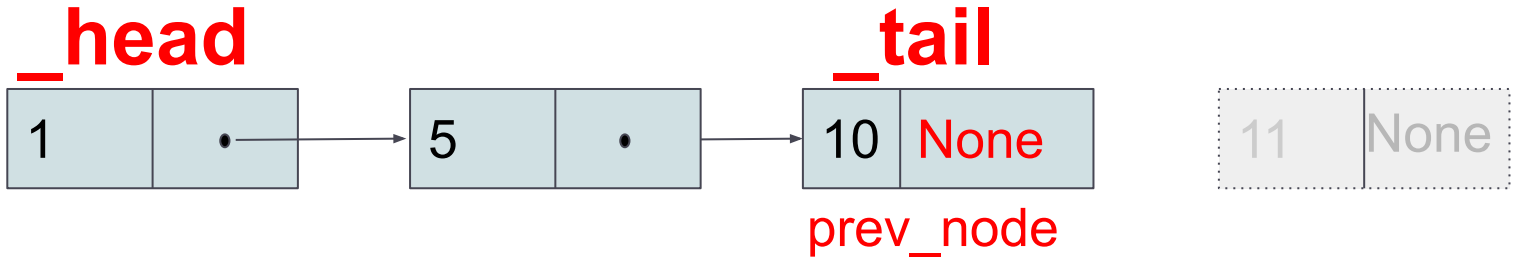

```
prev_node=self._head
while prev_node.next != self._tail :
    prev_node = prev_node.next
```

Implementación TAD Lista (_head y _tail)

- Cuando ya estamos en el penúltimo nodo, solo necesitamos que su atributo next sea **None** y actualizar _tail para sea al nuevo último nodo.

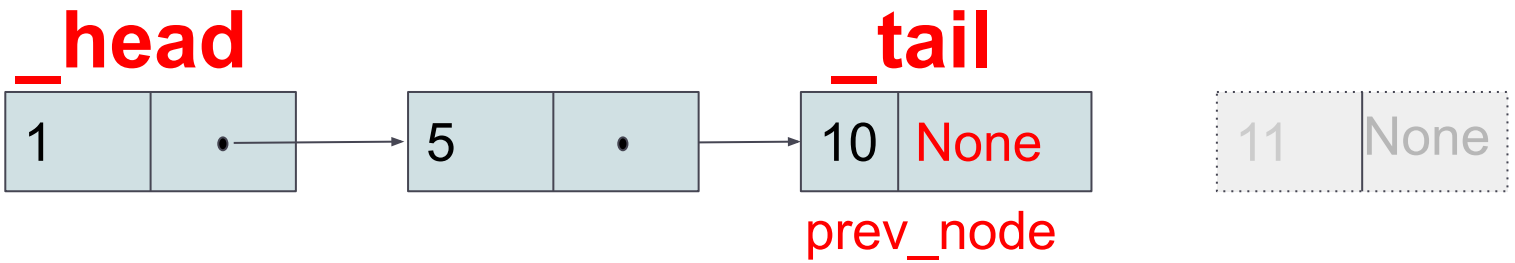



```
prev_node.next = None
tail = prev_node
```



Implementación TAD Lista (_head y _tail)

- El **recolector de basura** se ocupará de liberar el espacio de memoria ocupado por el antiguo último nodo (elemento 11), cuando detecte que no está siendo utilizado por ningún proceso.



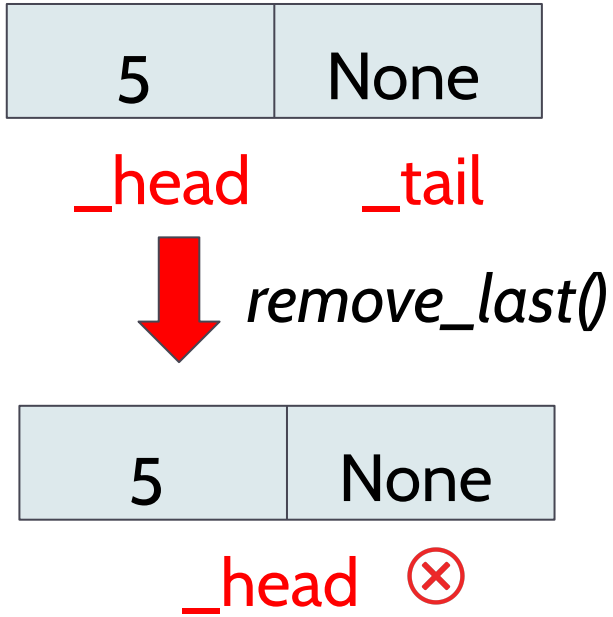

```
prev_node.next = None
_tail = prev_node
```

Implementación TAD Lista (_head y _tail)

- Para que tu solución sea robusta, debe verificar que es correcto para todos los posibles casos, por ejemplo:
  - lista vacía
  - lista con varios elementos
  - lista con un único elemento
- Debes comprobar que el resto de métodos siguen funcionando correctamente.
- Por ejemplo, en remove_last(), es necesario modificar _head en algún caso?.

Implementación TAD Lista (_head y _tail)

- Si nuestra lista tiene un único elemento cuando llamamos a remove_last(), ocurre lo siguiente: _head seguirá apuntando al nodo, cuando en realidad también debería ser None. Por tanto, tenemos que contemplar ese caso.

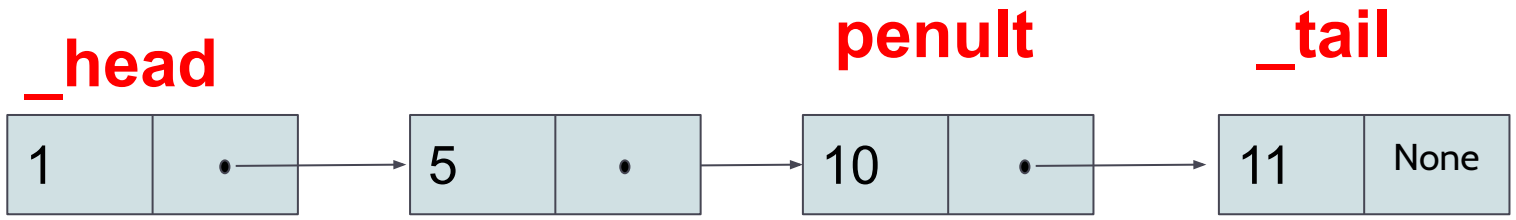


result = 5 ✔

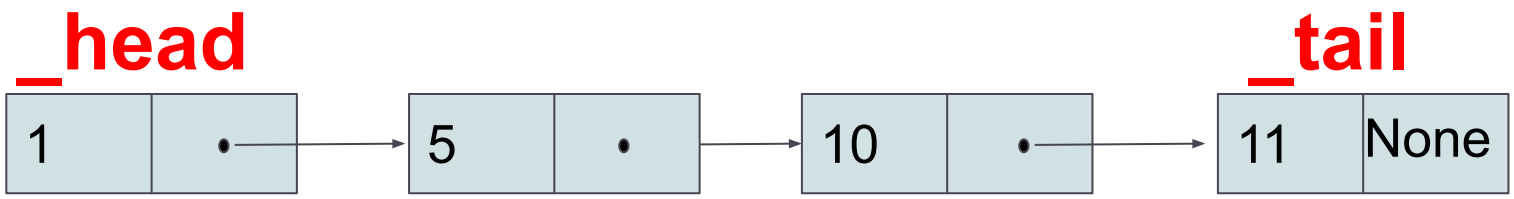
_tail = None ✔

Análisis de complejidad de remove_last()

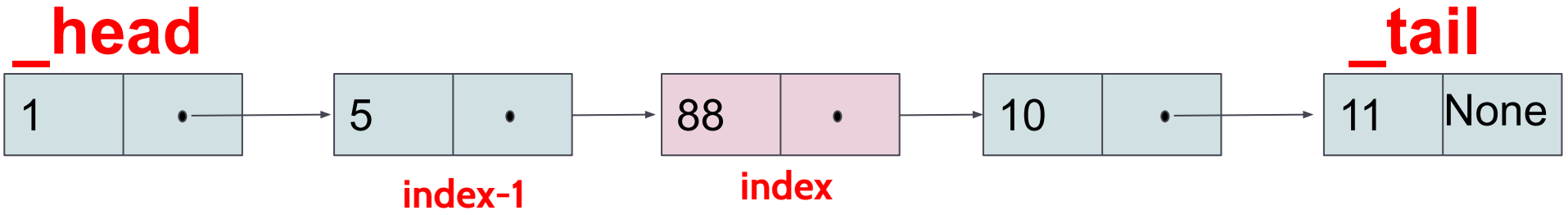
- ¿El uso de _tail en remove_last() mejora la eficiencia del método?.
- No, porque sigue siendo necesario recorrer la lista hasta alcanzar el penúltimo nodo.



Implementación TAD Lista (_head y _tail)

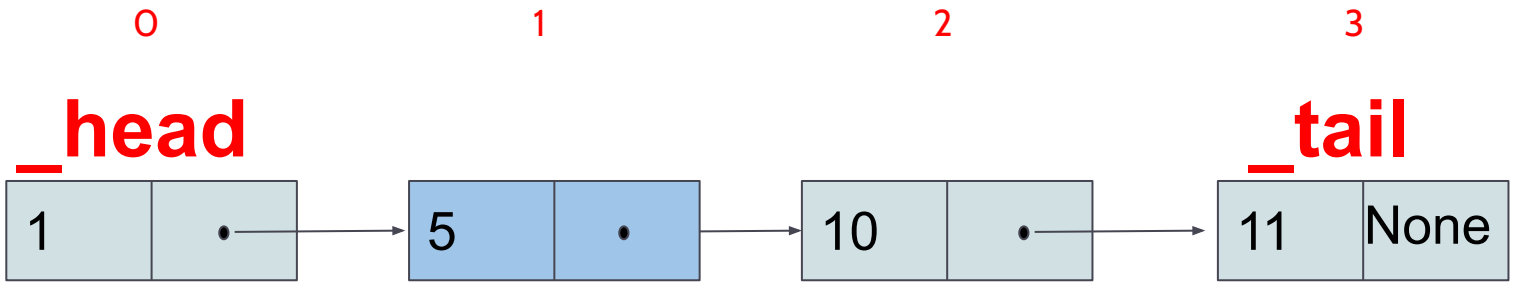


insertAt(2,88)



Implementación TAD Lista (_head y _tail)

- Debemos iterar hasta alcanzar el nodo en la posición index - 1.

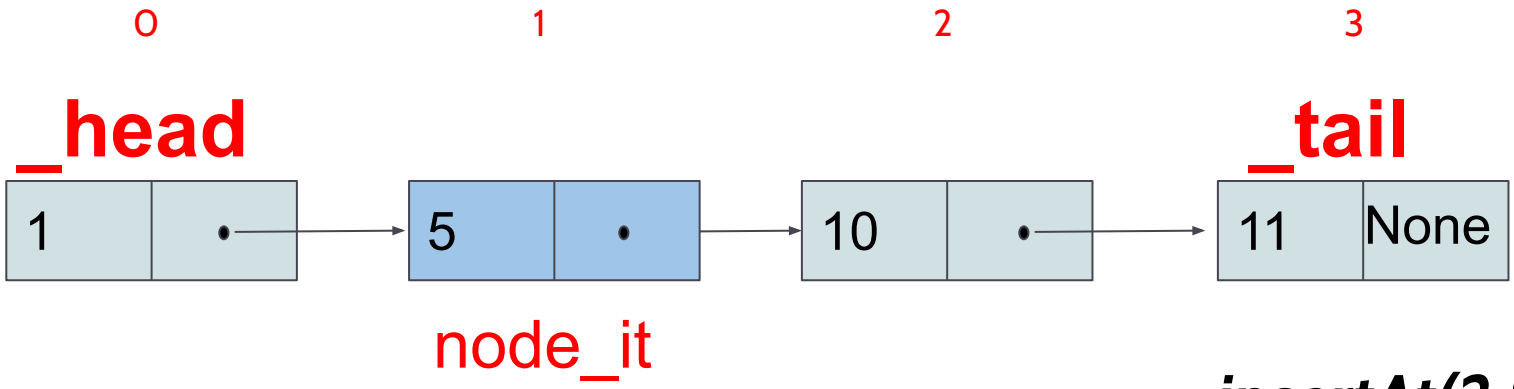


insertAt(2,88)

```
node_it = _head
```

Implementación TAD Lista (_head y _tail)

- Debemos iterar hasta alcanzar el nodo en la posición index - 1.

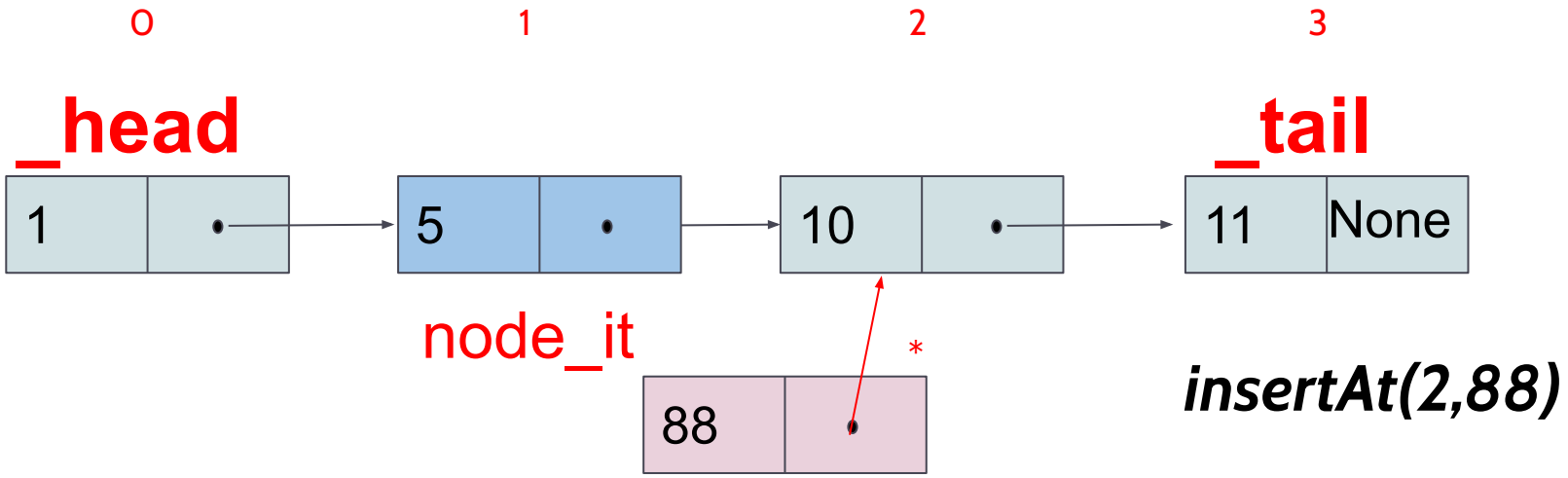


insertAt(2,88)

```
node_it = _head
for _ in range(index-1):
    node_it = node_it.next
# node_it es el nodo en la posición index -1
```

Implementación TAD Lista (_head y _tail)

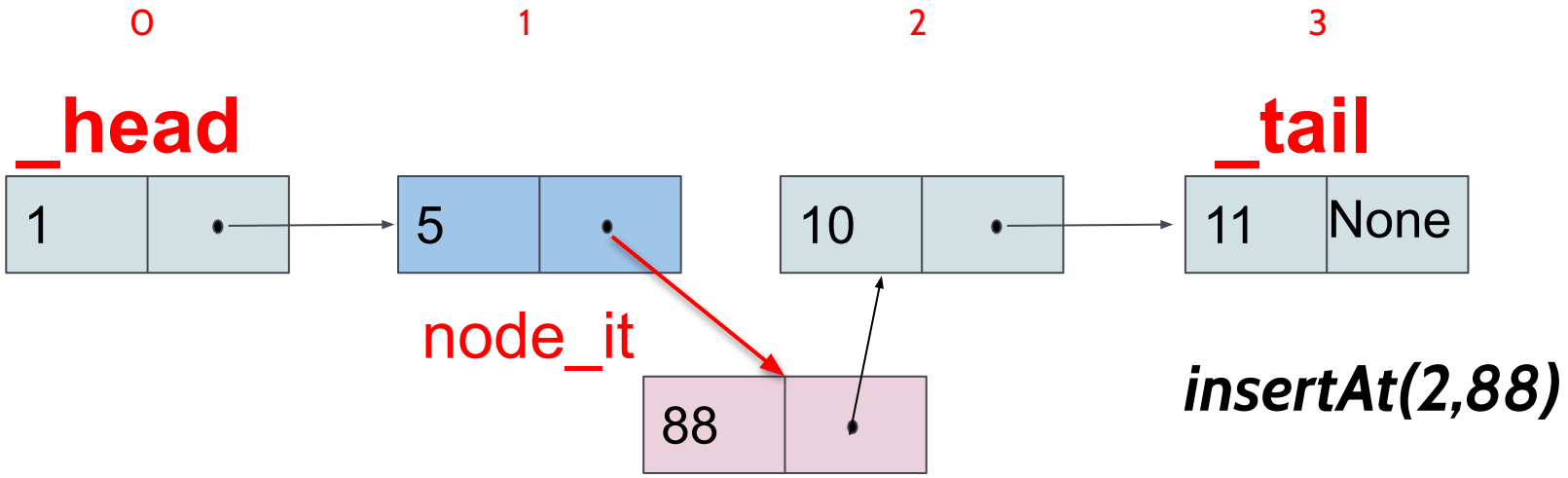
- Creamos el nuevo nodo y que apunte al siguiente a node_it




```
new_node = SNode(88, node_it.next)
```

Implementación TAD Lista (_head y _tail)

- Por último, simplemente tenemos que hacer que node_it.next apunte al nuevo nodo



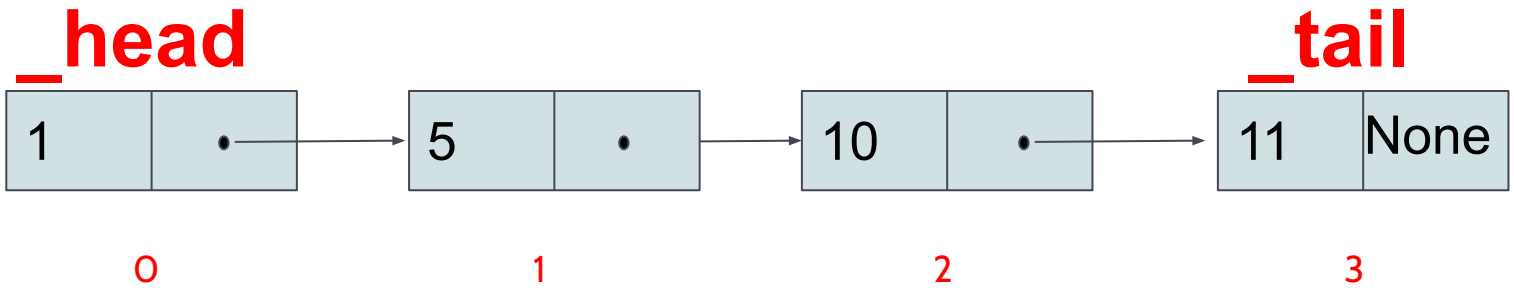

```
new_node = SNode(88, node_it.next)
node_it.next = new_node
```

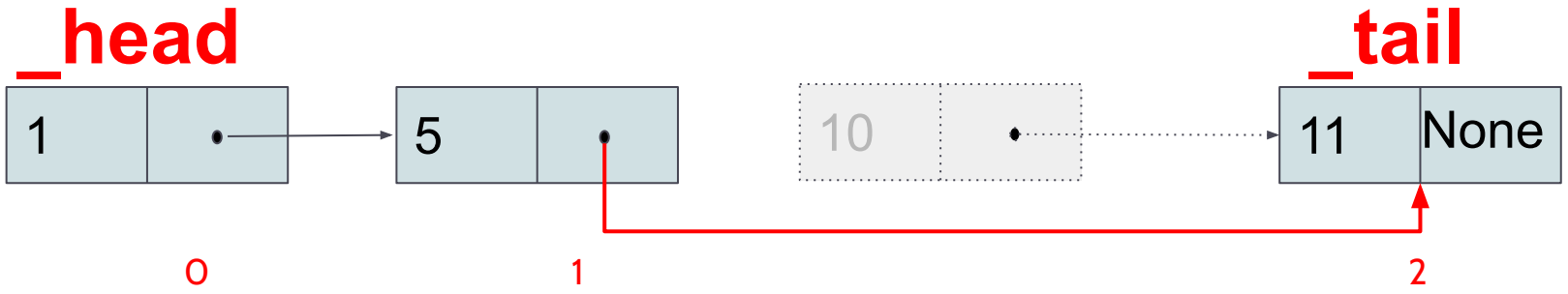
Implementación TAD Lista (_head y _tail)

- Recuerda que es imprescindible comprobar que tu implementación de **insertAt** es **robusta**, es decir, que funciona correctamente para distintos tipos de entrada:
  - índice fuera de rango
  - lista vacía
  - lista con un único elemento
  - lista con varios elementos
- Debes comprobar que el resto de métodos siguen funcionando correctamente.
- Recuerda también la importancia de la refactorización. Es recomendable re-utilizar métodos cuando sea posible.

Implementación TAD Lista (_head y _tail)

removeAt(index) (por ejemplo, removeAt(2))

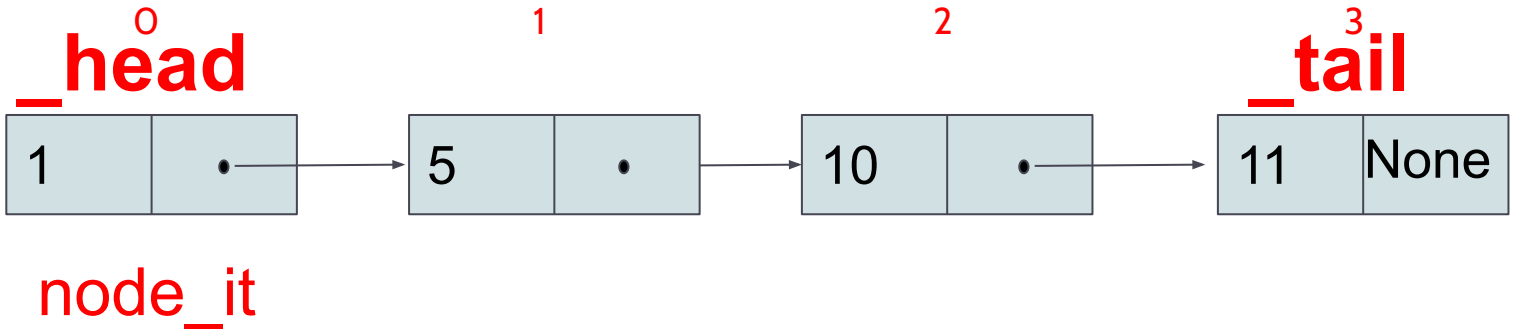




Implementación TAD Lista (_head y _tail)

- Debemos iterar hasta alcanzar el nodo en la posición index-1

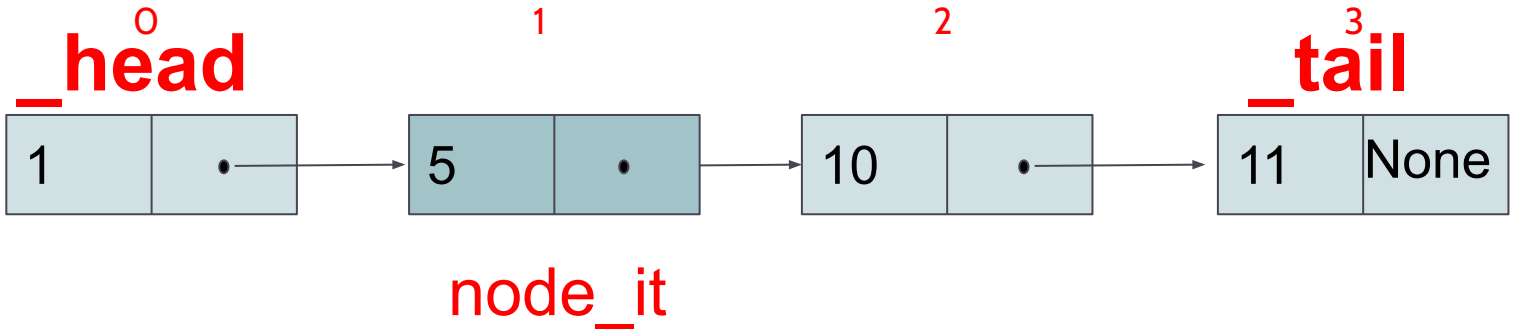
removeAt(2)



Implementación TAD Lista (_head y _tail)

- Debemos iterar hasta alcanzar el nodo en la posición index-1

removeAt(2)

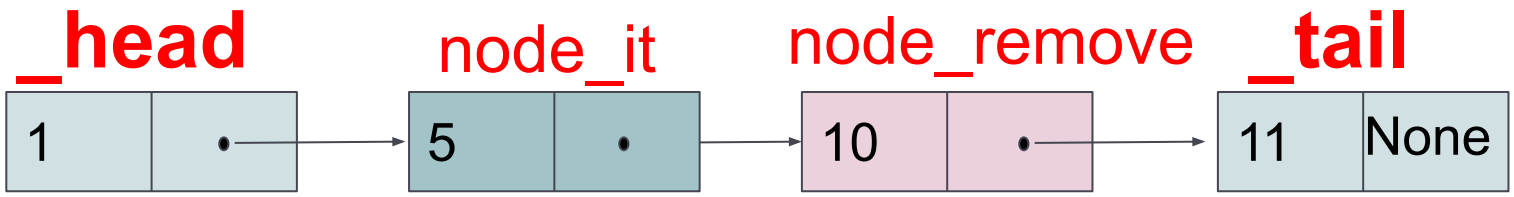



```
node_it = _head
for _ in range(index-1):
    node_it = node_it.next
# node_it es el nodo en la posición index -1
```

Implementación TAD Lista (_head y _tail)

- Usamos result para almacenar el nodo del elemento a eliminar

removeAt(2)

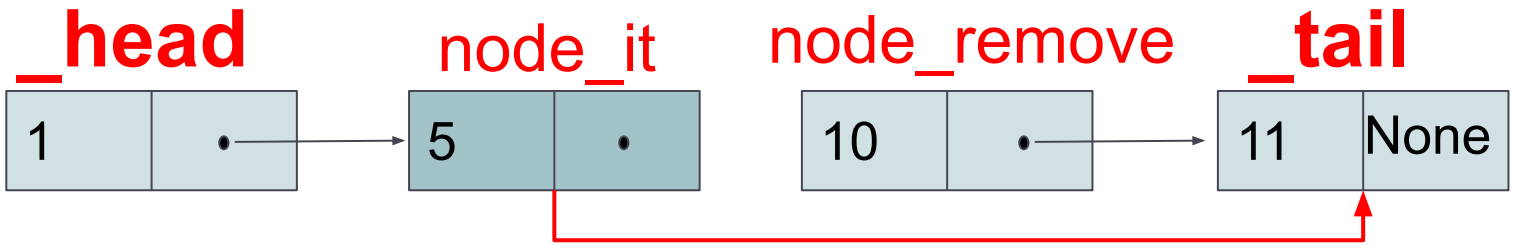



```
nodo_remove = node_it.next
result = node_remove.elem      # result = 10



return result
```

Implementación TAD Lista (_head y _tail)

- Falta que node_it.next apunte al siguiente al nodo eliminado

removeAt(2)

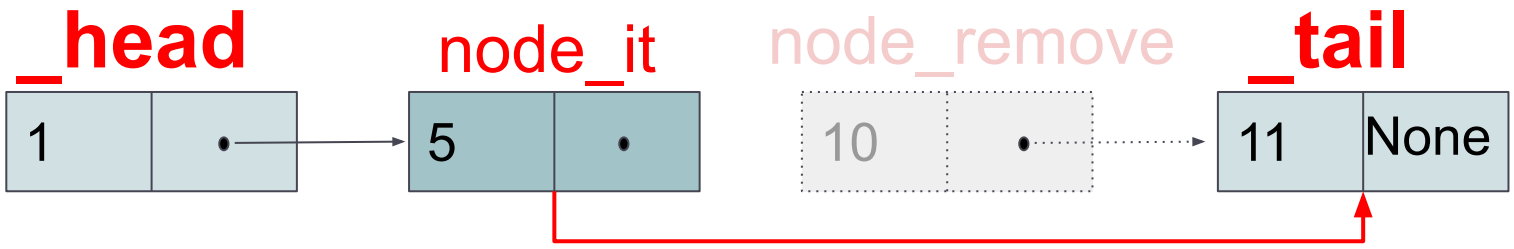



```
nodo_remove = node_it.next
result = node_remove.elem      # result = 10
node_it.next = node_remove.next
return result
```

Implementación TAD Lista (_head y _tail)

- El recolector de basura se encargará de liberar las direcciones de memoría que no están siendo utilizadas.

removeAt(2)




```
nodo_remove = node_it.next
result = node_remove.elem      # result = 10
node_it.next = node_remove.next
return result
```

Implementación TAD Lista (_head y _tail)

- Recuerda que es imprescindible comprobar que tu implementación de **removeAt** es robusta, es decir, que funciona correctamente para distintos tipos de entrada:
  - índice fuera de rango
  - lista vacía
  - lista con un único elemento
  - lista con varios elementos
- Debes comprobar que el resto de métodos siguen funcionando correctamente.
- Recuerda también la importancia de la refactorización. Es recomendable re-utilizar métodos cuando sea posible.

Ejercicio

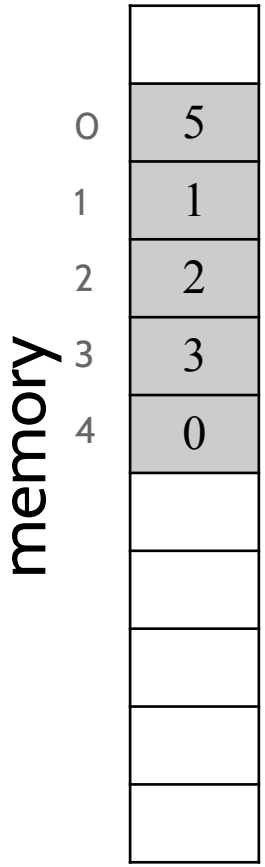
- Implementa el resto de métodos de la clase SList con _head y _tail. Solución
- Implementa la clase SList únicamente con _head. Solución
- ¿Qué ventajas aporta el uso del atributo _size?
  - Método len más eficiente.
- ¿Qué ventajas aporta el uso de _tail?.
  - Únicamente mejora la eficiencia de add_last

Índice

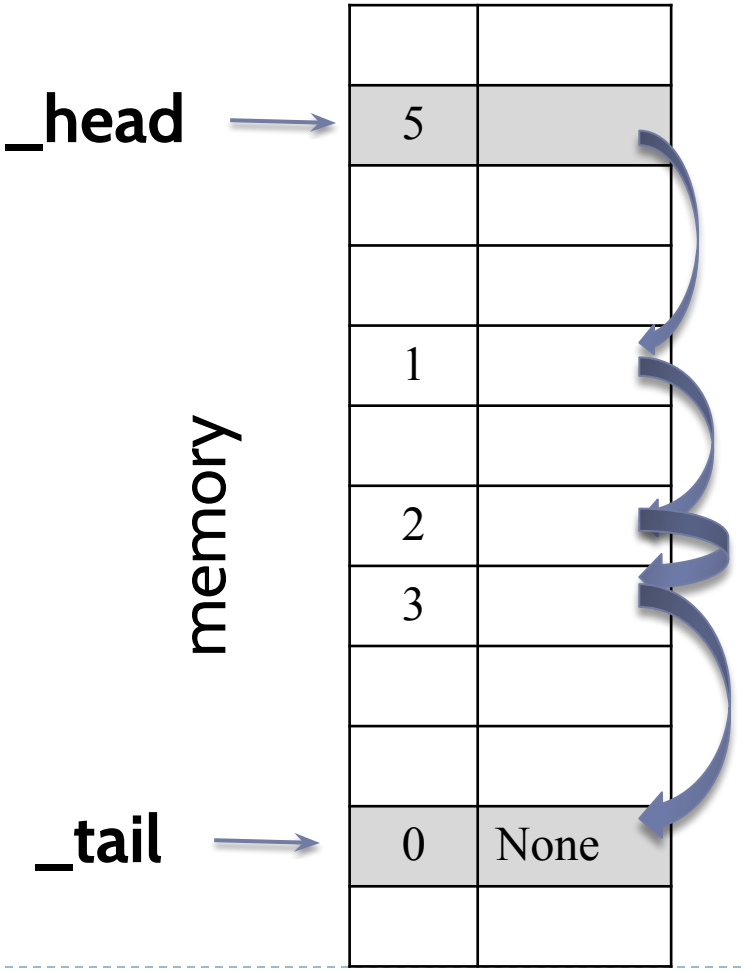
- Lista de Python (array)
- Lista Enlazada
  - Nodo Simple
  - Lista Simplemente Enlazada
- Implementación del TAD Pila basada en lista enlazada.
- Implementación del TAD Cola basada en lista enlazada.
- Definición TAD Lista.
- Implementación TAD Lista usando lista simplemente enlazada.
- **Implementación TAD Lista usando lista simplemente enlazada.**

Array vs Lista Simplemente Enlazada

Array (Python List)



Lista Simplemente Enlazada



Array vs Lista Simplemente Enlazada

Array (Lista de Python)

(+) Rápido y fácil acceso a los elementos.

(-) Las operaciones de Inserción y borrado son lentas (porque es necesario mover elementos a otras posiciones de memoria).

Lista Simplemente Enlazada

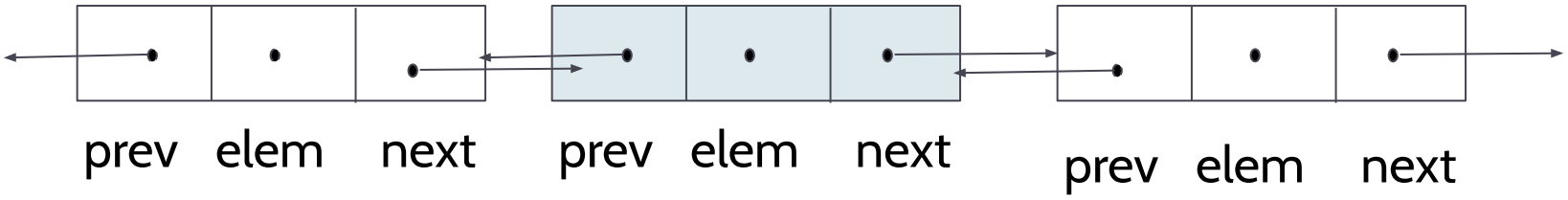
(-) Necesita más espacio

(-) Acceso secuencial (debes visitar todos los elementos anteriores a uno dado).

(+) Las operaciones de inserción y borrado son más eficientes.

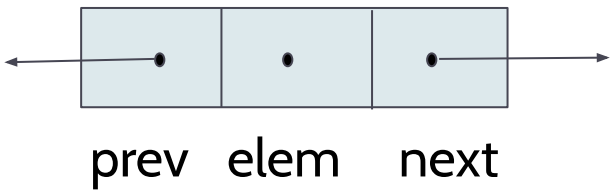
¿Cómo mejorar el acceso a los nodos?

- Además de la referencia **next**, cada nodo tiene otra referencia que apunta al nodo anterior (**prev**) en la lista.
- Es posible visitar la lista de izquierda a derecha, y también en orden inverso.



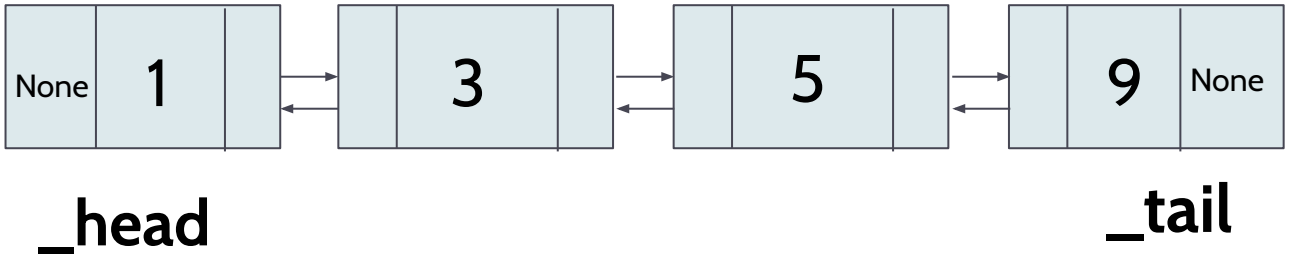
Nodo doblemente enlazado

```
class DNode:
    def __init__(self, e: object, prev_node: 'DNode' = None, next_node: 'DNode' = None) -> None:
        self.elem = e
        self.next = next_node
        self.prev = prev_node
```



Lista doblemente enlazada

- Usa dos referencias:
  - _head: primer nodo de la lista.
  - _tail: último nodo de la lista.



Constructor

- El constructor crea una lista vacía. head and tail deben ser None.

```
class DList:
    def __init__(self) -> None:
        """creates an empty list"""
        self._head = None
        self._tail = None
        self._size = 0
```

TAD Lista

- Si es necesario, revisa la definición y operaciones del TAD Lista (slide 59).

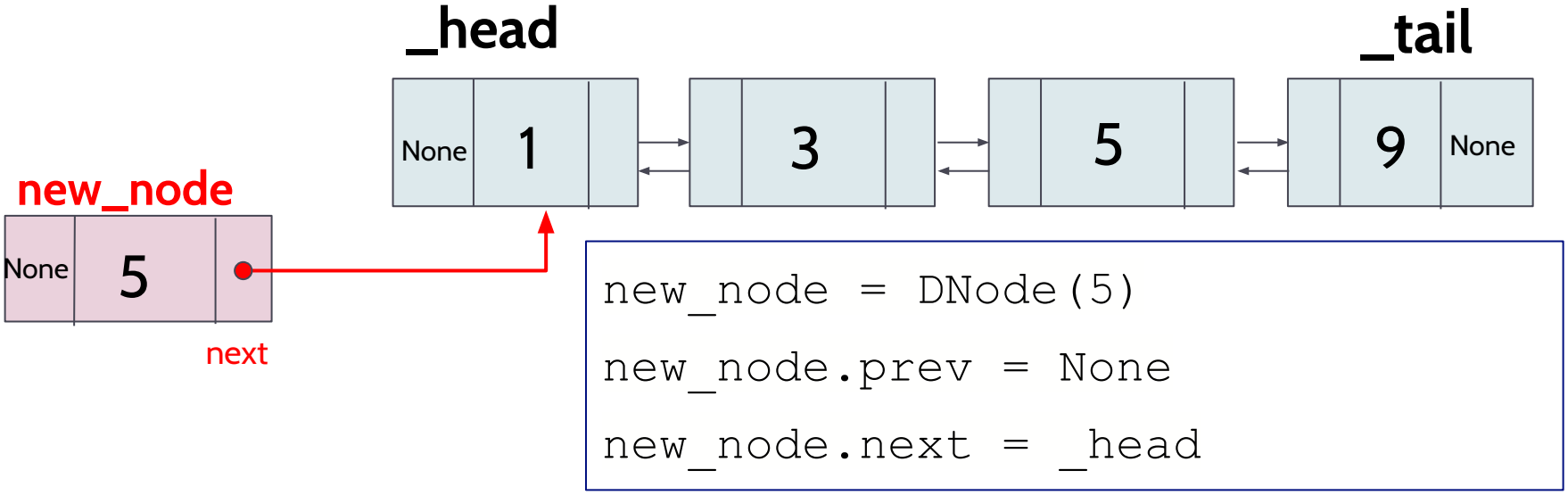
Implementación TAD Lista con lista doblemente enlazada

- **add_first(e) añade el elemento e al principio de la lista.**
- add_last(e) añade el elemento e al final de la lista.
- remove_first() borra el primer elemento de la lista.
- remove_last() borra el último elemento de la lista.

Implementación TAD Lista con lista doblemente enlazada

Algoritmo add_first (por ejemplo, l.addFirst(5)):

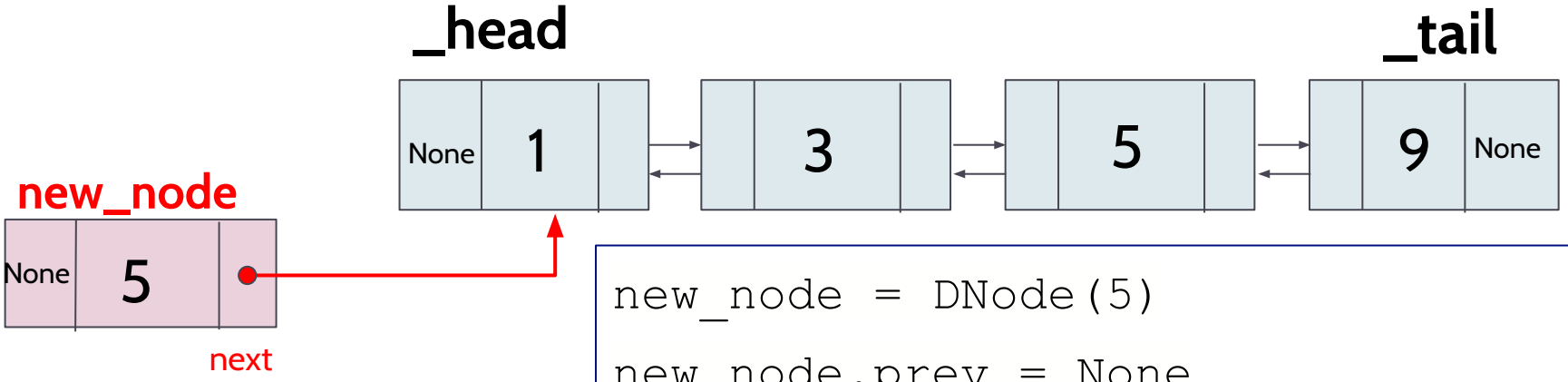
1) **Crear un nuevo nodo, cuyo siguiente sea _head.**



Implementación TAD Lista con lista doblemente enlazada

Algoritmo add_first (por ejemplo, l.addFirst(5)):

1) **Crear un nuevo nodo, cuyo siguiente sea _head.**



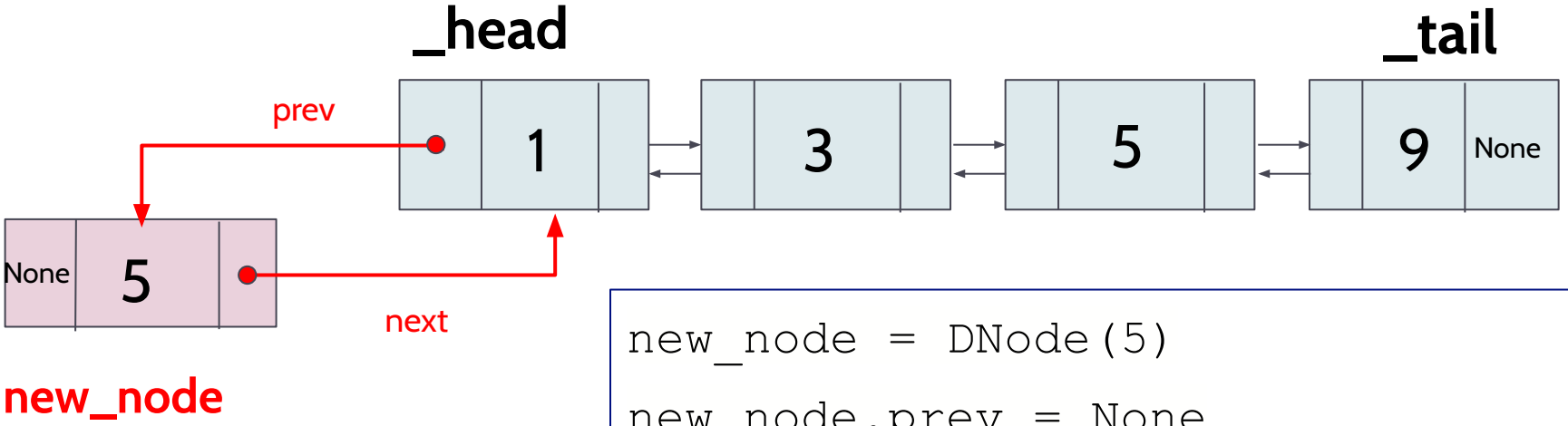

```
new_node = DNode(5)
new_node.prev = None
new_node.next = _head
```

Como el constructor de DNode admite argumentos para el nodo previo y siguiente, una alternativa es:

```
new_node = DNode(5, None,_head)
```

Implementación TAD Lista con lista doblemente enlazada

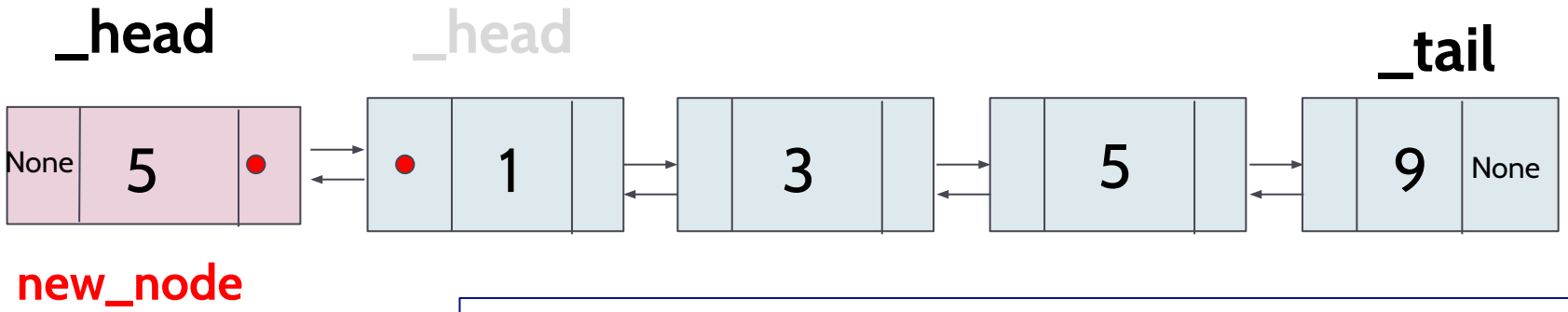
Algoritmo add_first (por ejemplo, l.addFirst(5)):
2) _head.prev también debe estar conectado al nuevo nodo




```
new_node = DNode(5)
new_node.prev = None
new_node.next = _head
_head.prev = new_node
```

Implementación TAD Lista con lista doblemente enlazada

Algoritmo add_first (por ejemplo, l.addFirst(5)):
3) Tenemos que **modificar la referencia head, ye incrementar size.**



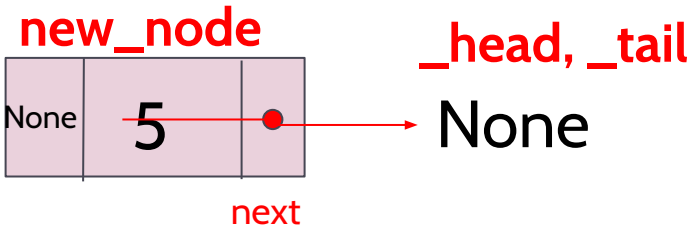

```
new_node = DNode(5)
new_node.prev = None
new_node.next = _head
_head.prev = new_node
_head = new_node
_size += 1
```

Implementación TAD Lista con lista doblemente enlazada)

- La solución implementada es correcta cuando la lista tiene varios elementos, pero debemos asegurarnos que nuestra solución es robusta.
- Para que tu solución sea robusta, debe verificar que es correcta para cualquier tipo de lista:
  - lista vacía
  - lista con un único elemento
- Además, es muy importante que compruebes que el resto de métodos siguen funcionando correctamente.

Implementación TAD Lista con lista doblemente enlazada)

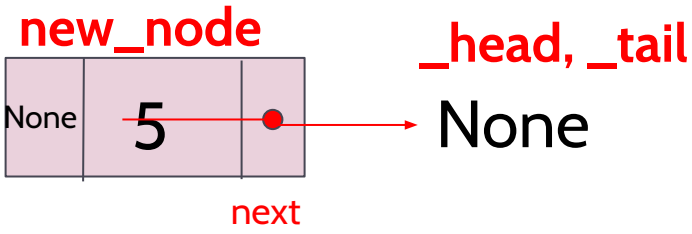
- **add_first() cuando la lista está vacía:**
  - Con la creación del nodo no hay problema. El nodo con .next debe apuntar a _head, que es None.




```
new_node = DNode(5)
new_node.prev = None
new_node.next = _head
_head.prev = new_node
_head = new_node
_size += 1
```

Implementación TAD Lista con lista doblemente enlazada)

- Sin embargo, la instrucción _head.prev = new_node, lanza una excepción porque _head es None, y None no tiene ningún atributo o método.



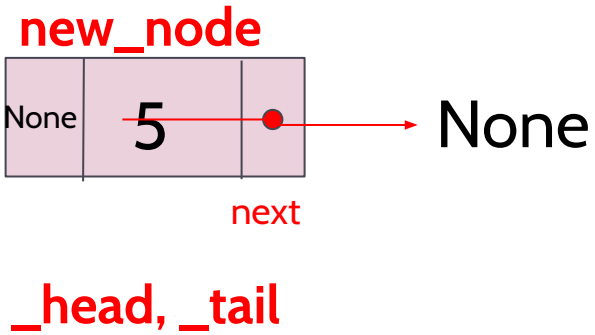

```
new_node = DNode(5)
new_node.prev = None
new_node.next = _head
_head.prev = new_node
_head = new_node
_size += 1
```

Error!!!

Implementación TAD Lista con lista doblemente enlazada)

- Debes modificar el código para que controle el caso en el que la lista está vacía

Solución (ver add_first en DList)




```
new_node = DNode(5)
new_node.prev = None
new_node.next = _head

if _head is None:
    _tail = new_node
else:
    _head.prev = new_node
_head = new_node
_size += 1
```

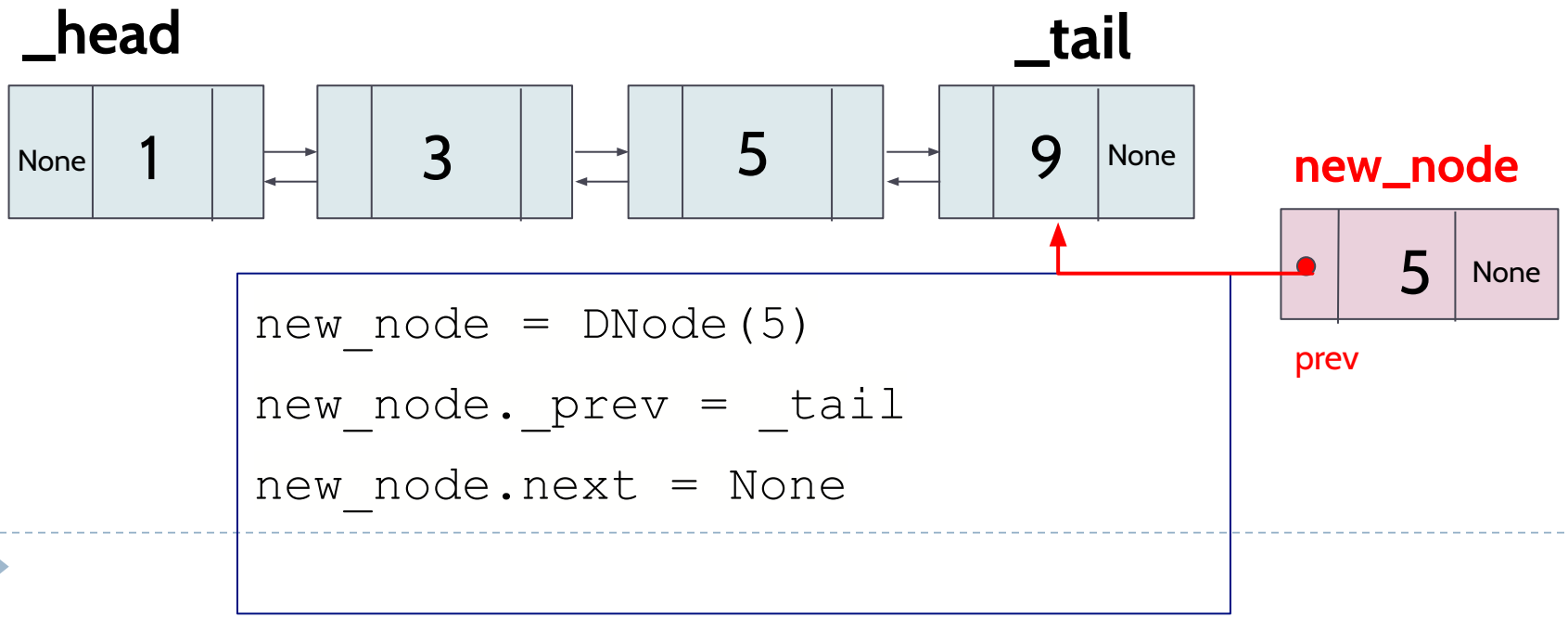
Implementación TAD Lista con lista doblemente enlazada

- add_first(e) añade el elemento e al principio de la lista.
- **add_last(e) añade el elemento e al final de la lista.**
- remove_first() borra el primer elemento de la lista.
- remove_last() borra el último elemento de la lista.

Implementación TAD Lista con lista doblemente enlazada

Algoritmo add_last (por ejemplo, l.add_last(5)):

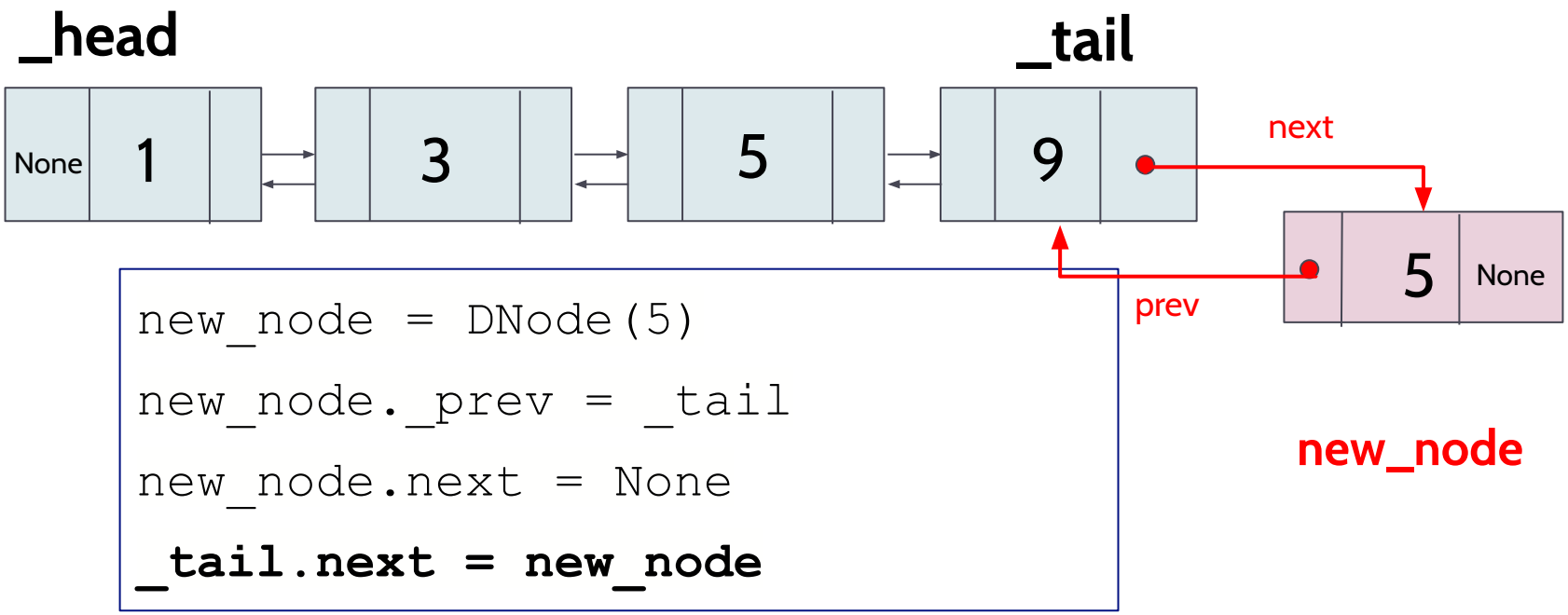
1) **Crear un nuevo nodo, y enlazalo con _tail.**
2) Modificar la referencia _tail.



Implementación TAD Lista con lista doblemente enlazada

Algoritmo add_last (por ejemplo, l.add_last(5)):

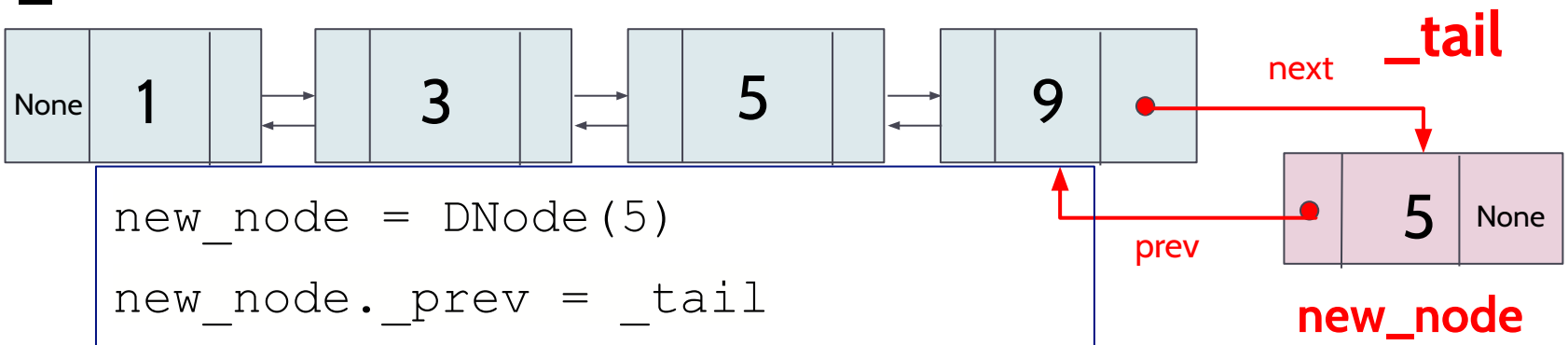
1) **Crear un nuevo nodo, y enlazalo con _tail.**
2) Modificar la referencia _tail.



Implementación TAD Lista con lista doblemente enlazada

Algoritmo add_last (por ejemplo, l.add_last(5)):

1) Crear un nuevo nodo, cuyo siguiente sea _tail, y enlazalo con tail
2) **Modificar la referencia _tail, e incrementa size.**




```
new_node = DNode(5)
new_node._prev = _tail
new_node.next = None
_tail.next = new_node
_tail = new_node
_size += 1
```

Implementación TAD Lista con lista doblemente enlazada)

- Para que tu solución sea robusta, debe verificar que es correcto para todos los posibles casos, por ejemplo:
  - lista con varios elementos
  - lista con un único elemento
  - lista vacía
- Debes comprobar que el resto de métodos siguen funcionando correctamente.
- Por ejemplo, en **add_last**(), es necesario modificar _head en algún caso?.
  - Sí, cuando la lista está vacía, ambos _head y _tail deben ser actualizados para que sean el nuevo nodo que se acaba de añadir.
- Solución.

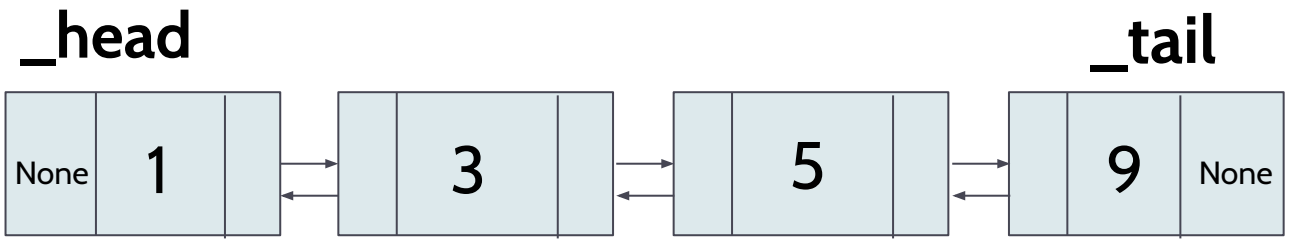
Implementación TAD Lista con lista doblemente enlazada

- add_first(e) añade el elemento e al principio de la lista.
- add_last(e) añade el elemento e al final de la lista.
- **remove_first() borra el primer elemento de la lista.**
- remove_last() borra el último elemento de la lista.

Implementación TAD Lista con lista doblemente enlazada

Algoritmo remove_first:

1) **Almacena el valor del primer nodo en una variable.**
2) Actualizar _head

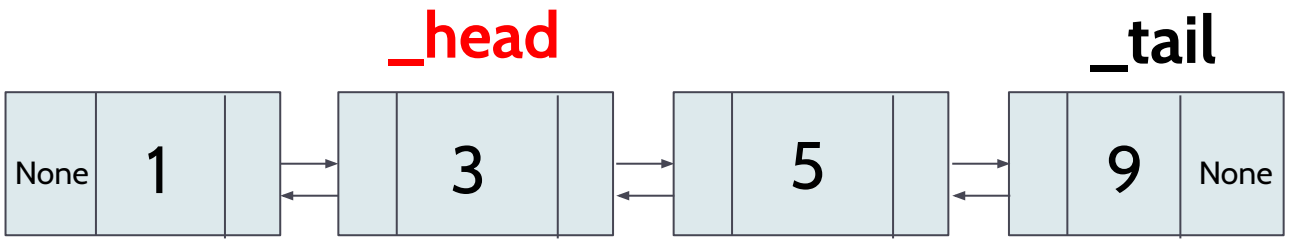



```
result = _head.elem    # result = 1
…
return result
```

Implementación TAD Lista con lista doblemente enlazada

Algoritmo remove_first:

1) Almacena el valor del primer nodo en una variable.
2) **Actualizar _head**

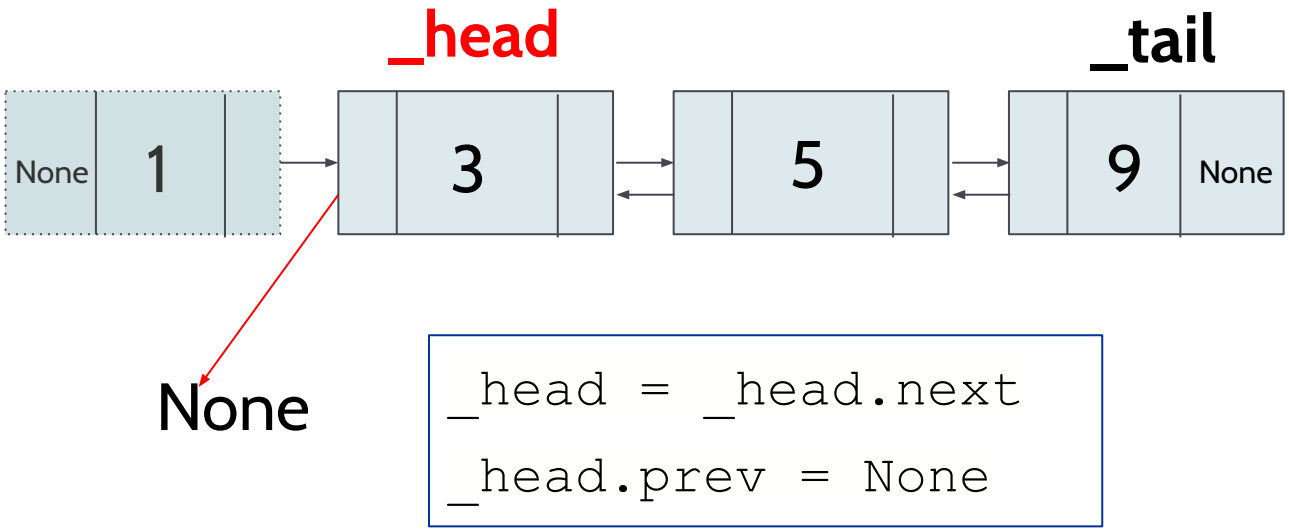



```
result = _head.elem    # result = 1
_head = _head.next
return result
```

Implementación TAD Lista con lista doblemente enlazada

Algoritmo remove_first:

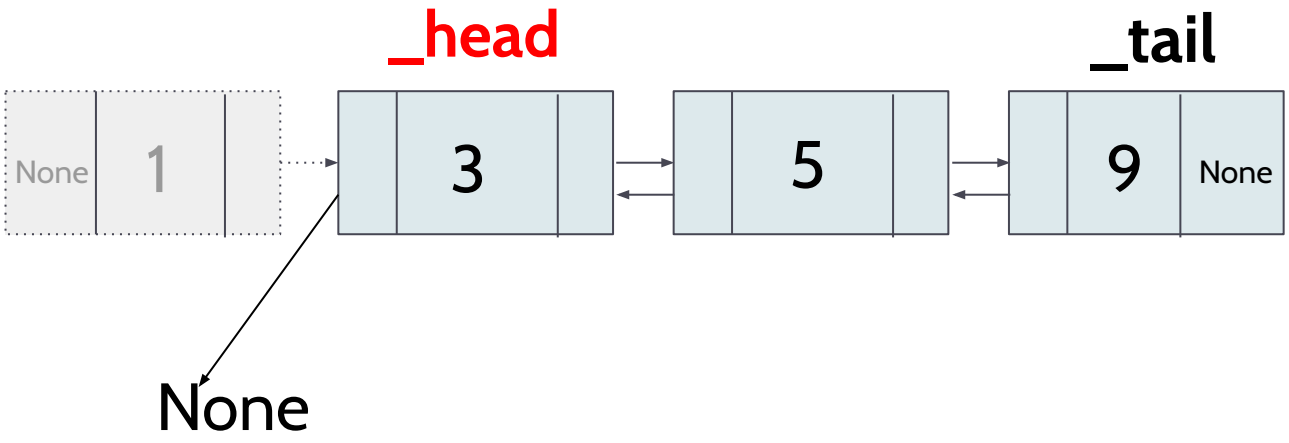
1) Almacena el valor del primer nodo en una variable.
2) Actualizar _head. Además, debemos hacer que _head.prev sea None.



Implementación TAD Lista con lista doblemente enlazada

Algoritmo remove_first:

- El recolector de basura se ocupará de liberar el espacio de memoria que ocupan los objetos que no están siendo utilizados ni referenciados.

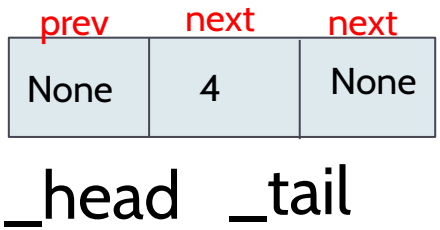


Implementación TAD Lista con lista doblemente enlazada)

- Para que tu solución sea robusta, debe verificar que es correcto para todos los posibles casos, por ejemplo:
  - lista con varios elementos
  - lista vacía
  - lista con un único elemento
- Debes comprobar que el resto de métodos siguen funcionando correctamente.
- Solución.

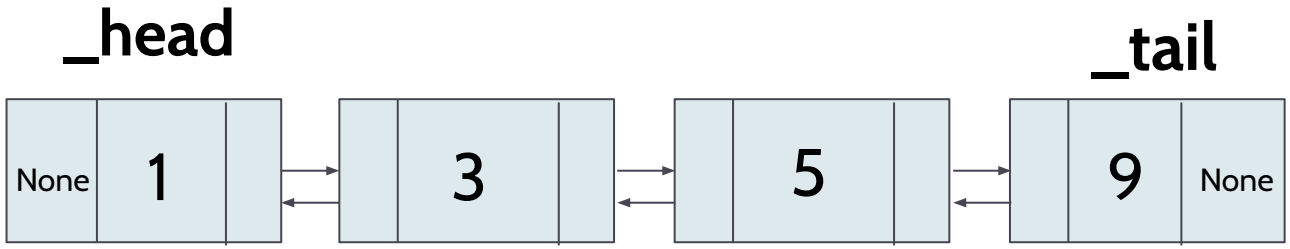
Implementación TAD Lista con lista doblemente enlazada)

- En una lista con un único elemento, tras ejecutar **remove_first**(), _head será None, pero además, necesitamos asegurarnos que _tail también pase a ser None



Implementación TAD Lista con lista doblemente enlazada)

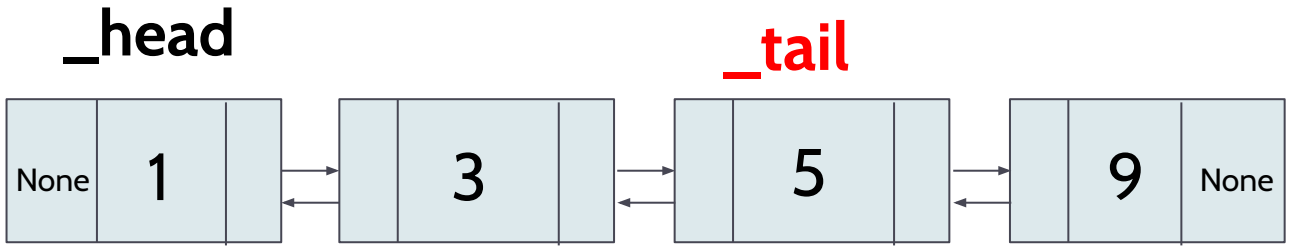
1) **Almacenar el elemento en una variable**
2) Modificar la referencia _tail y _tail.next debe ser None
3) Devolver resultado.




```
result = _tail.elem #9
```

Implementación TAD Lista con lista doblemente enlazada)

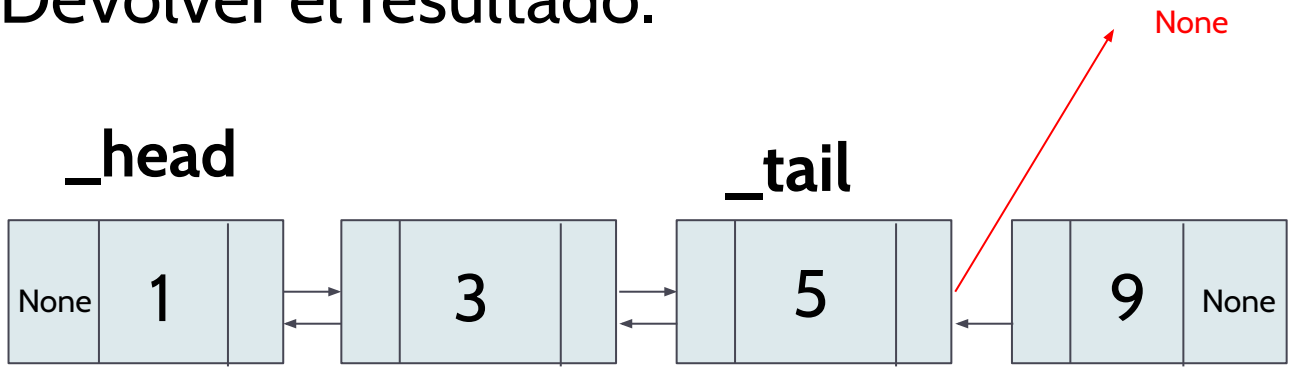
1) Almacenar el elemento en una variable
2) **Modificar la referencia _tail** y _tail.next debe ser None
3) Devolver el resultado.




```
result = _tail.elem #9
_tail = _tail.prev
```

Implementación TAD Lista con lista doblemente enlazada)

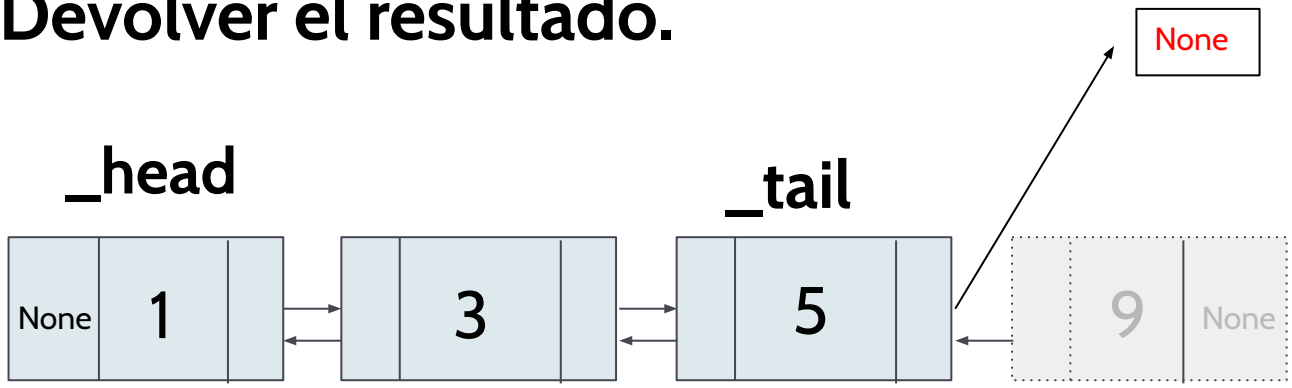
1) Almacenar el elemento en una variable
2) Modificar la referencia _tail y **_tail.next debe ser None**
3) Devolver el resultado.




```
result = _tail.elem #9
_tail = _tail.prev
_tail.next = None
```

Implementación TAD Lista con lista doblemente enlazada)

1) Almacenar el elemento en una variable
2) Modificar la referencia _tail y _tail.next debe ser None
3) **Devolver el resultado.**




```
result = _tail.elem #9
_tail = _tail.prev
_tail.next = None
return result
```

Implementación TAD Lista con lista doblemente enlazada)

- Para que tu solución sea robusta, debe verificar que es correcto para todos los posibles casos, por ejemplo:
  - lista con varios elementos
  - lista vacía
  - lista con un único elemento
- Debes comprobar que el resto de métodos siguen funcionando correctamente.
- Solución.

Implementación TAD Lista con lista doblemente enlazada)

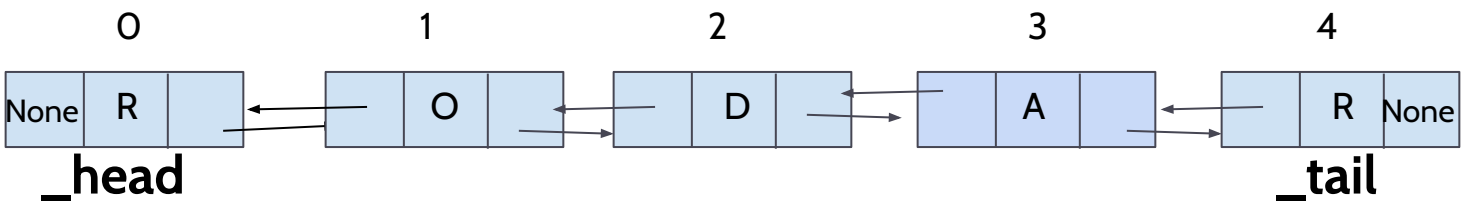
- Completa el resto de los métodos de la clase DList.
- Solución

Problema: Palabras palíndromas

- Una palabra palíndroma es una palabra que se lee igual en ambos sentidos. Por ejemplo: *radar*, *anna, level, civic, madam, noon.*
- Implementa una función Python que reciba una palabra y devuelve True si es palíndroma, y False en otro caso.
- Tu solución tiene que usar una lista doblemente enlazada. Cada nodo contiene un único carácter de la palabra.

Problema: Palabras palíndromas

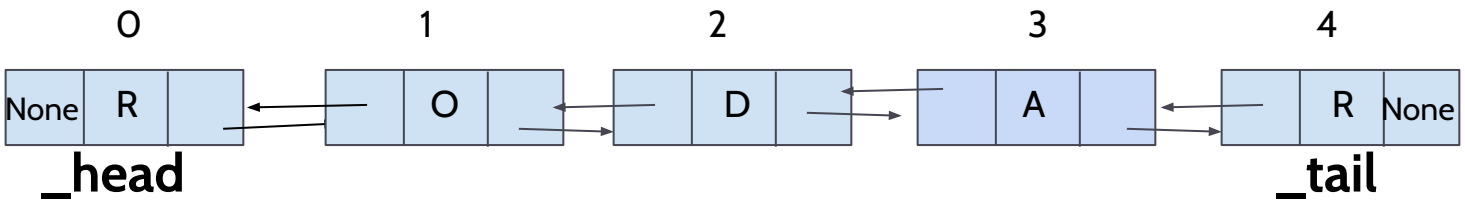
```
def is_palindrome(word: str):
    ...
```



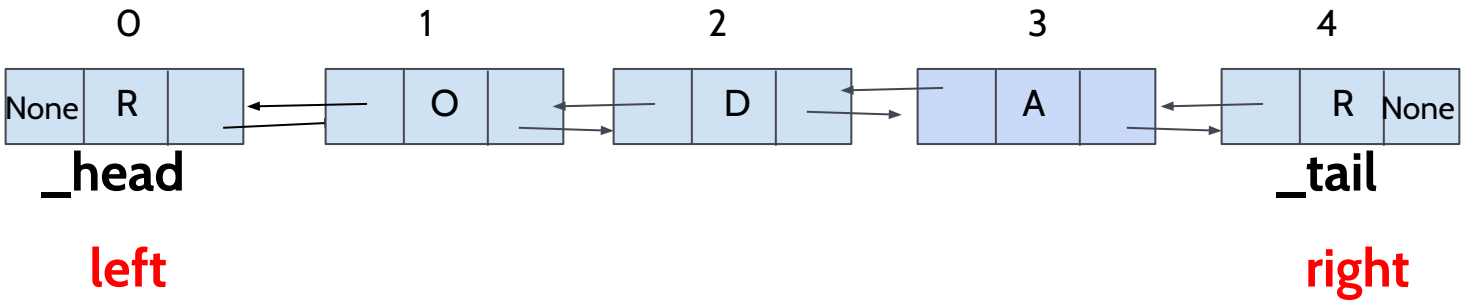
word='RODAR'

Problema: Palabras palíndromas

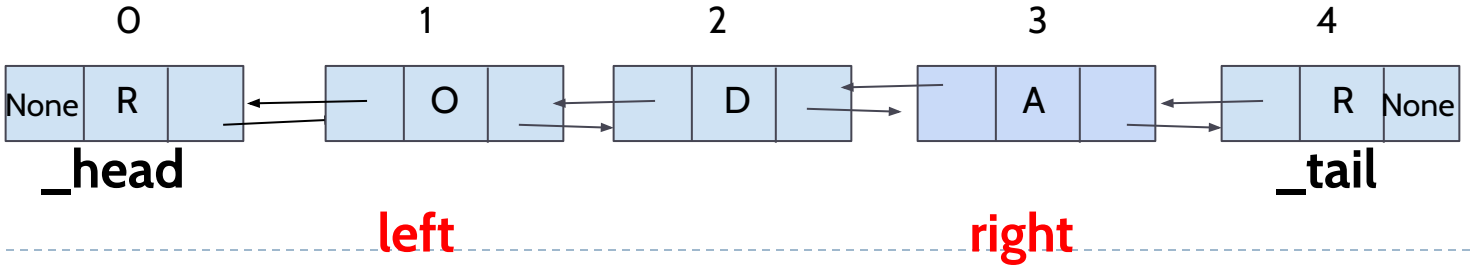
```
def is_palindrome(word:str):
    result=False
    l=DList()
    for c in word:
        l.addLast(c)
    …
    return result
```



Problema: Palabras palíndromas

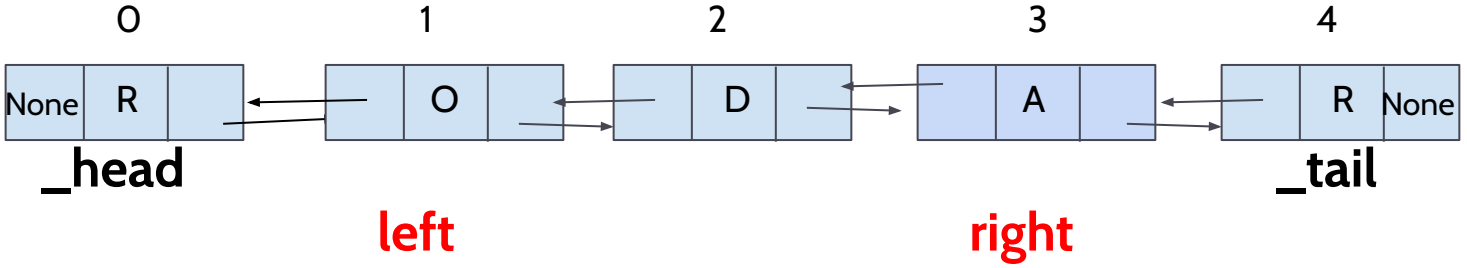


mientras left y right tengan el mismo elemento (left.elem == right.elem), podremos avanzar:
left = left.next
right = right.prev



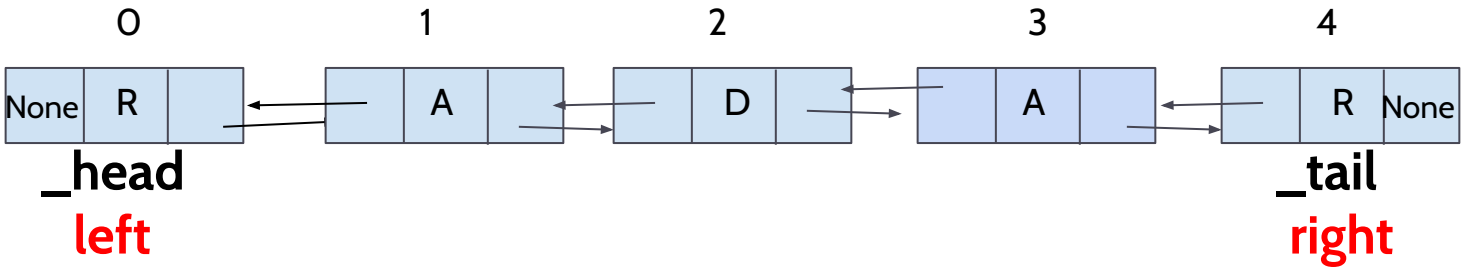
Problema: Palabras palíndromas

En el momento, que left y right contienen elementos distintos podemos afirmar que la palabra no es palíndroma, y devolver False
right = right.prev



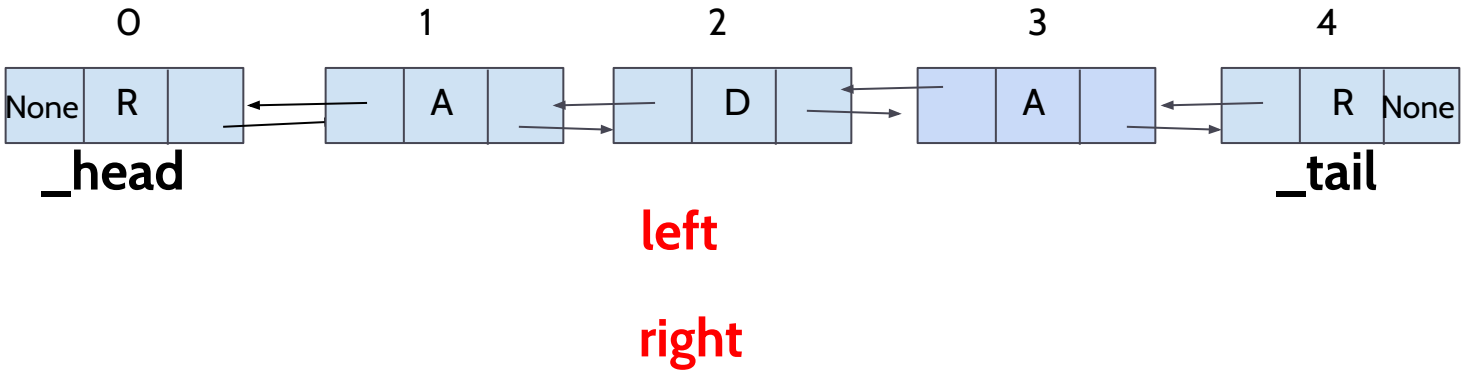
Problema: Palabras palíndromas

Probemos ahora con una palabra palíndroma: radar



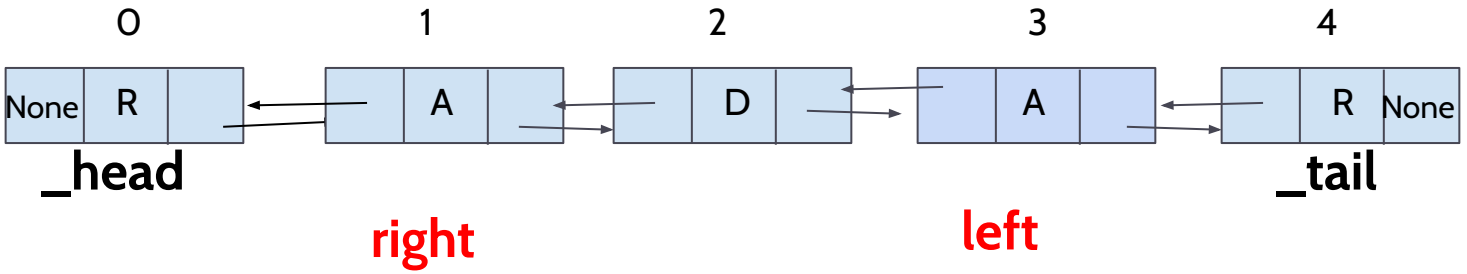
Problema: Palabras palíndromas

Probemos ahora con una palabra palíndroma: radar



Problema: Palabras palíndromas

¿Es necesario seguir comprobando?



Problema: Palabras palíndromas

- Implementa tu solución comprobando que tu función es correcta y robusta en al menos los siguientes casos:
  - palabras no palíndroma (distintas longitudes)
  - palabra palíndroma de tamaño par
  - palabra palíndroma de tamaño impar

Nota: Para None, puedes considerar que no es palíndroma. Para una palabra de tamaño 0, puedes considerar que sí es Palíndroma.

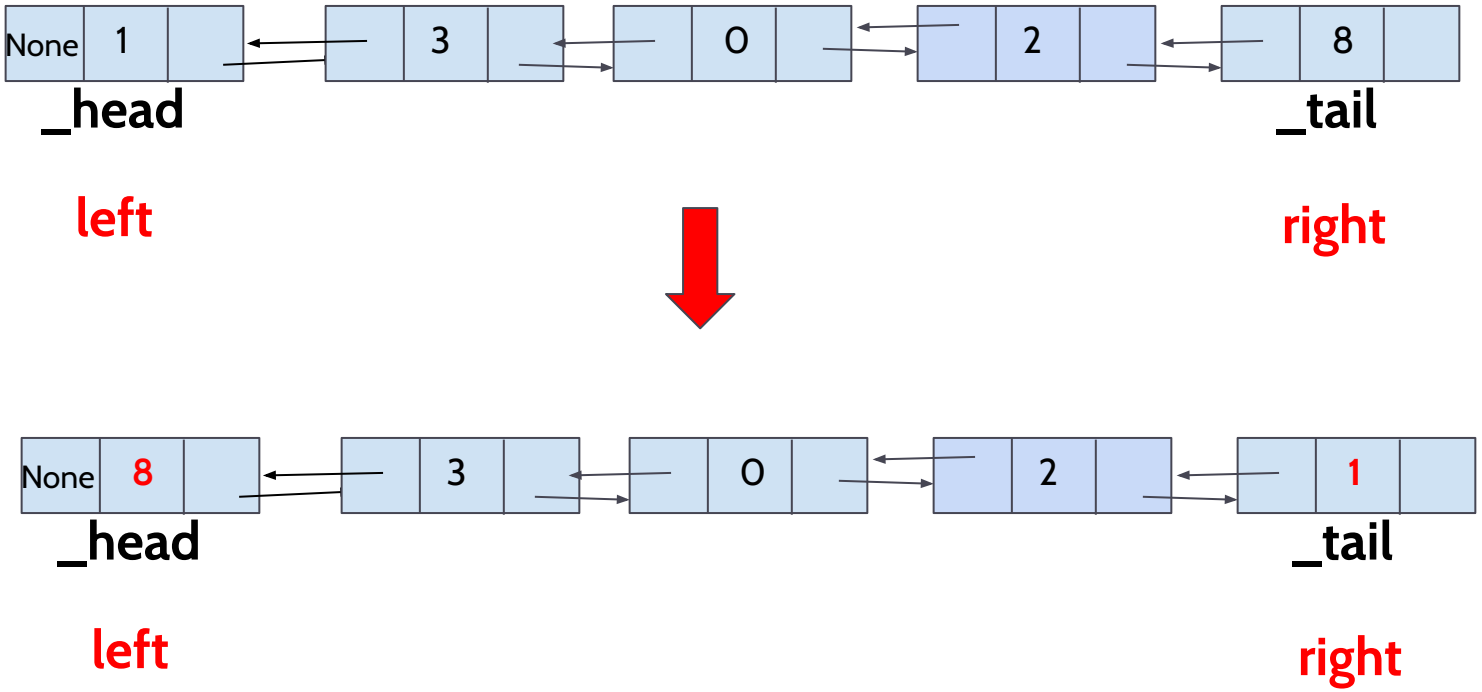
Solución

Problema: Invertir una lista doble

- En la clase DList, implementa un nuevo método que invierta la lista. una nueva función en la clase DList que invierta la lista. El método no devuelve una nueva lista, simplemente modifica la existente, invirtiendo el orden de sus elementos.
- Hay dos posibles enfoques:
  - Intercambiando los elementos de los nodos
  - Intercambiando los enlaces prev y next de cada nodo.

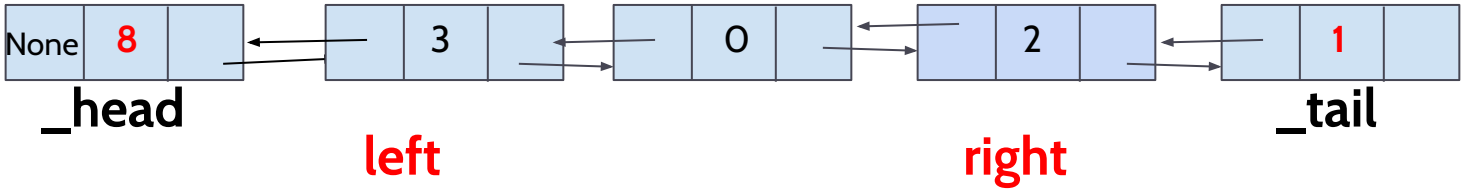
Problema: Invertir una lista doble

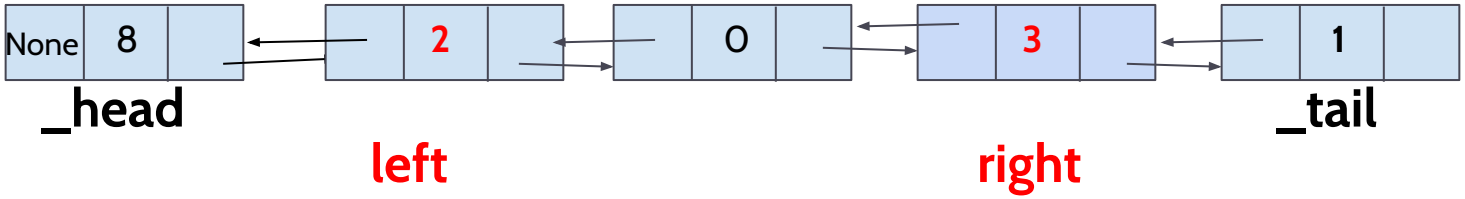
1) Intercambiar los elementos de los nodos



Problema: Invertir una lista doble

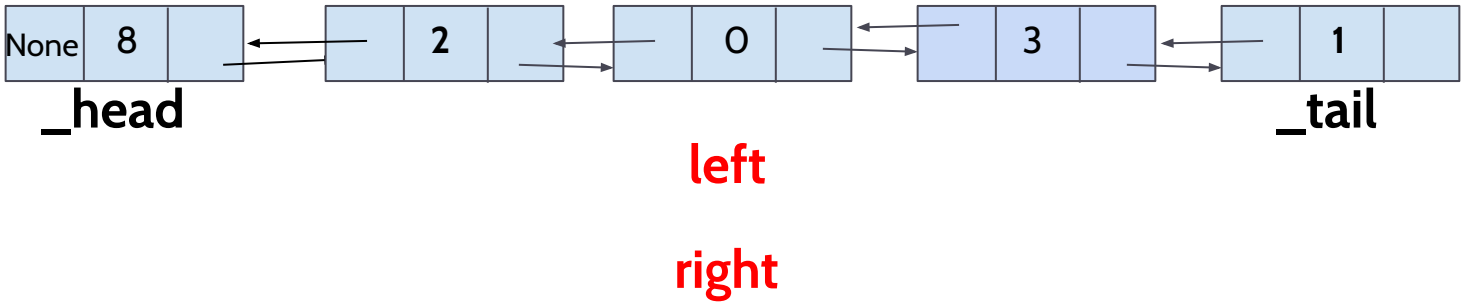
1) Intercambiar los elementos de los nodos





Problema: Invertir una lista doble

1) Intercambiar los elementos de los nodos



En este caso, paramos cuando left y right llegan al mismo nodo.
¿Qué pasa si la lista tiene un número impar de elementos?, ¿Qué condición podemos utilizar para saber que no tenemos que seguir iterando?

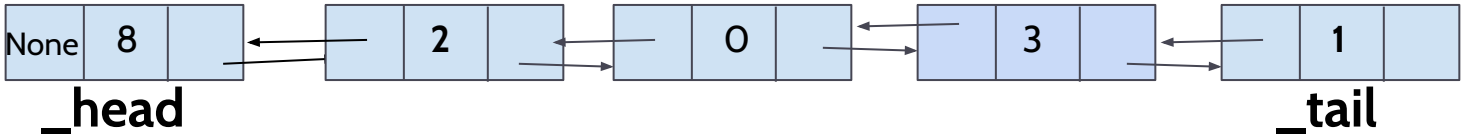
Problema: Invertir una lista doble

2) Para cada nodo, debemos intercambiar sus atributos next y prev.

old_next = node.next
node.next = node.prev
node.prev = old_next

Problema: Invertir una lista doble

2) Intercambiar los atributos next y prev para cada nodo. Comenzar por el primer nodo, y avanzar hasta _tail. Al final, tendréis que cambiar _head y _tail.



node

Solución

Resumen

- Las listas enlazadas permiten almacenar secuencias de elementos en memoria principal en direcciones de memoria no contiguas.
- Las listas enlazadas están formadas por nodos. Cada nodo almacena un elemento y la referencia al siguiente nodo en la lista.
- Este tipo de estructuras mejora la complejidad temporal de las operaciones de actualización: inserción y borrado, porque no es necesario mover el resto de elementos de la lista.

Resumen

- Sin embargo, el acceso a sus elementos siempre es secuencial, y por tanto, menos eficiente que el acceso a los elementos en un array.
- Los TAD Pilas y Colas pueden implementarse usando listas enlazadas.
- También hemos estudiado el TAD Lista y una implementación basada en lista enlazada.