Listas enlazadas

...

Algoritmos y Estructuras de Datos
2023

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

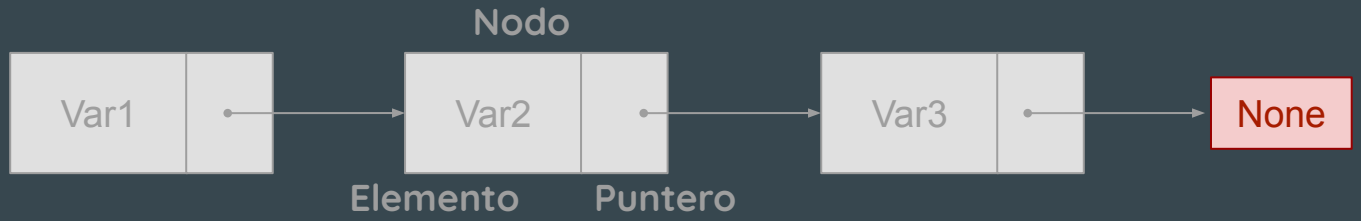
Añadir o quitar elementos en una lista es una operación muy costosa ¿Qué cambios en la estructura de datos podrían facilitar estos procesos?

Problemas con las lista de Python

- Las listas de Python son una clase sumamente optimizada y, usualmente, es la mejor opción para almacenar datos
- Sin embargo, posee algunas limitaciones
  - La memoria asignada al comienzo es siempre mayor al número de elementos para facilitar el agregado de valores en los extremos.
  - Agregar y quitar elementos en el interior de la lista es costoso
  - Sistemas con poca memoria o memoria muy fragmentada pueden encontrarse con problemas para encontrar espacio suficiente para almacenar la lista completa.

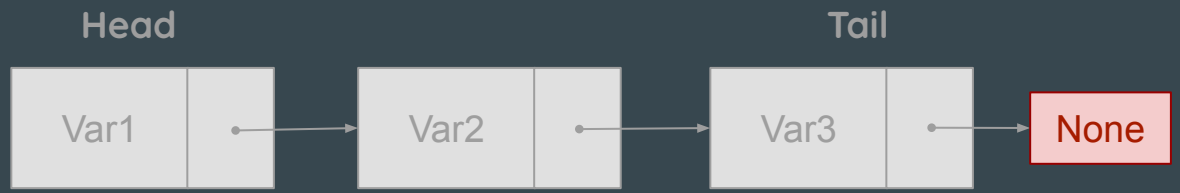
Listas enlazadas

- Los *arrays* responden a una representación centralizada de los datos: una gran porción de memoria capaz de almacenar todos los datos (y más).
- Las **listas enlazadas** se basan en una representación distribuida de los datos.
- Estas listas se conforman de **nodos** que almacenan un elemento de la lista y la dirección de memoria, un **puntero**, de uno o más elementos aledaños.



Listas simplemente enlazadas (LSE)

- Son la versión más simple de las listas enlazadas.
- Cada nodo posee dos valores: una referencia a un elemento de la secuencia y un puntero al nodo siguiente
- El primer elemento de la lista se conoce como **cabeza** (o ***head***) mientras que el último se denomina **cola** (o ***tail***).
- El elemento ***tail*** siempre apunta a un elemento nulo (***None*** en Python o ***Null*** en otros lenguajes)



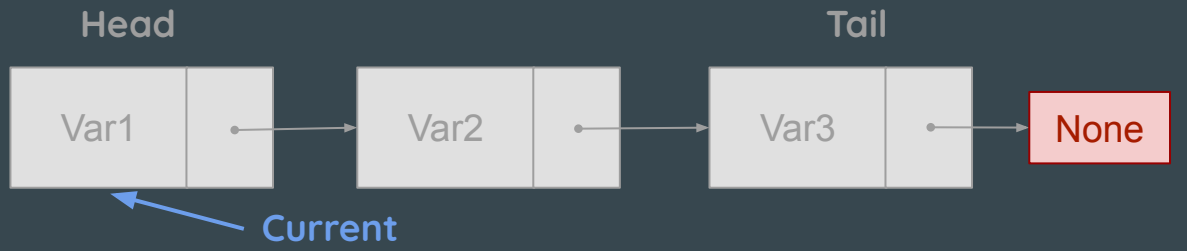
Implementando LSE

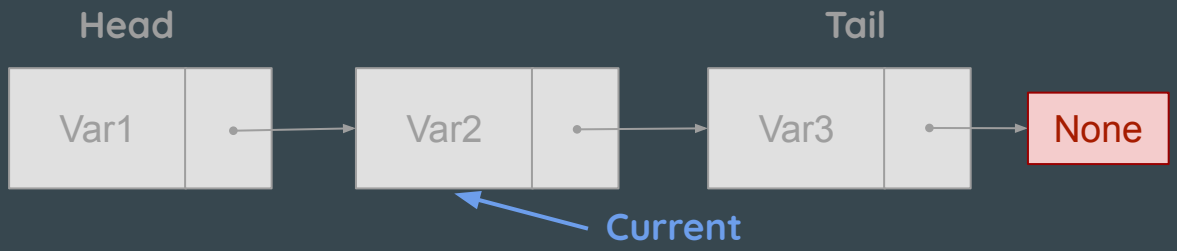
```
class Node:
    def __init__(self, e, n):
        self._element = e
        self._next = n

class LSE:
    def __init__(self):
        self._head = None
        self._len = 0

    def __len__(self):
        return self._len

    def empty(self):
        if self._len == 0:
            return True
```

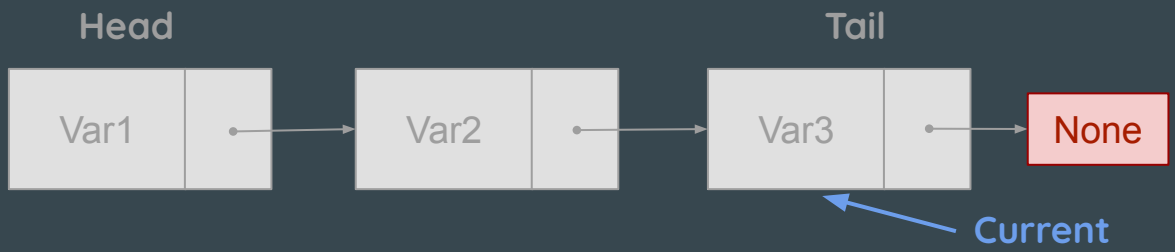
Recorrido de LSE

- Para recorrer una lista completa, siempre (y solo) se comienza desde el elemento ***head***.
- Moviéndose de un nodo al siguiente es posible recorrer toda la lista, hasta alcanzar el valor ***None***.
- Los distintos nodos de la lista se visitan usando el puntero ***current***.
- Se debe definir una función que mueva el puntero ***current*** a la posición siguiente.

Recorrido de LSE

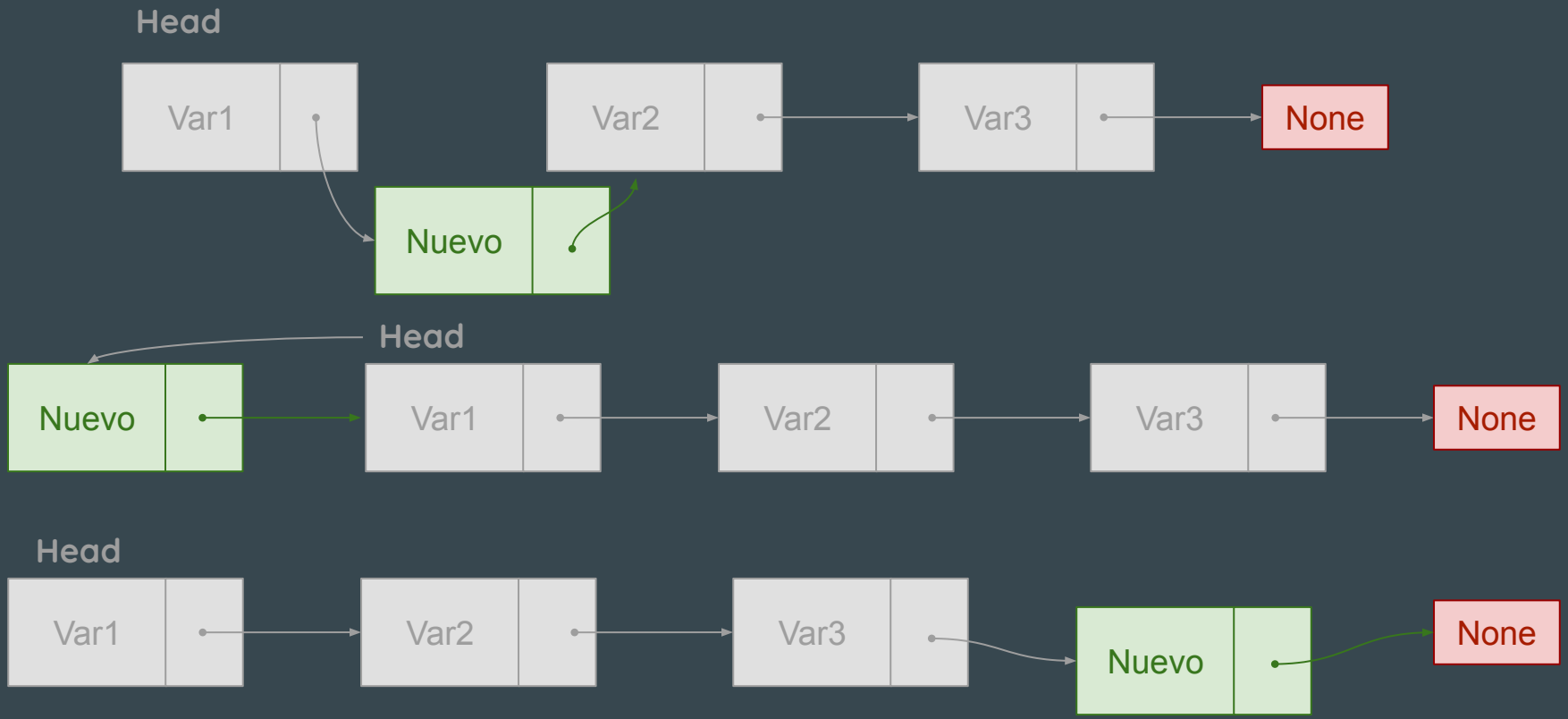




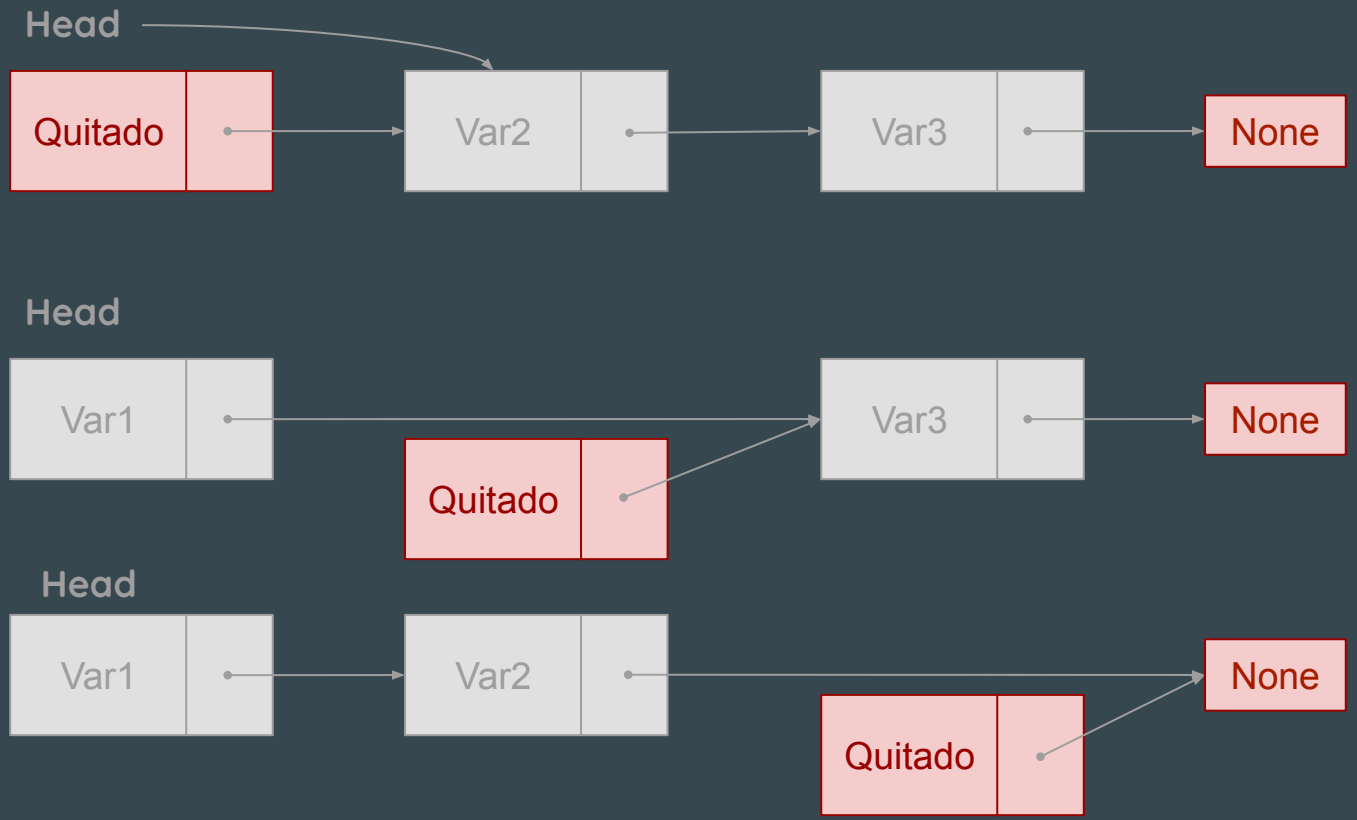



```
current = self._head
while current._next != None:
    current = current._next
```

Añadir un elemento a la LSE

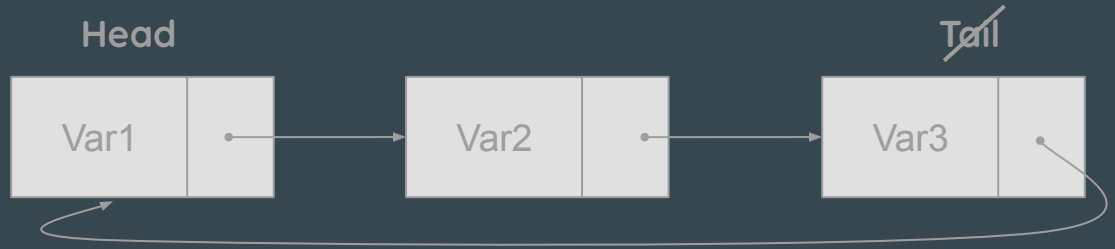


Quitar un elemento de la LSE



Listas circulares (LC)

- Son similares a las LSE, pero el último elemento apunta al nodo ***head***.
- Si bien no tienen un comienzo y fin como las LSE.
- Si bien las LC no poseen ni inicio ni fin estrictos, es necesario mantener una referencia a algún nodo para poder recorrer la lista.
- Este puntero puede llamarse ***head***

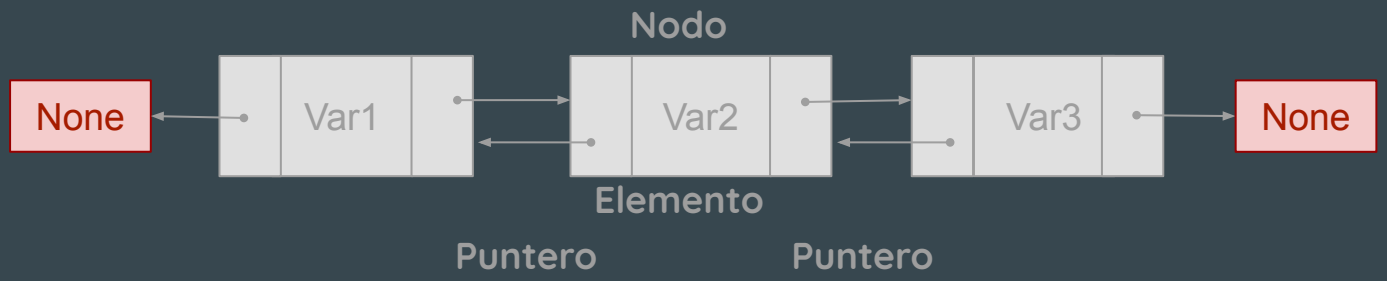


Trabajando con LC

- El recorrido es similar al de las LSE, pero se finaliza al llegar al primer elemento recorrido (nodo ***head***).
- Añadir o quitar elementos sigue la misma lógica que para las LSE, actualizando los nodos a los que apunta cada puntero.

Listas doblemente enlazadas (LDE)

- Las LSE no permiten el recorrido inverso de una lista y la eliminación de elementos en el interior de la lista es complejo.
- Para solucionar estos problemas es posible agregar un segundo puntero en cada nodo que apunte al elemento anterior en la lista.
- Estas estructuras se denominan **listas doblemente enlazadas**.



Implementando LDE

```
class Node:
     def __init__(self, e, n, p):
         self._element = e
         self._next = n
         self._prev = p

class LDE:
    def __init__(self):
        self._head = None
        self._tail = None
        self._len = 0

    def __len__(self):
        return self._len

    def empty(self):
        if self._len == 0:
            return True
```

```
def __getitem__(self, i):
     if i >= len(self):
          raise IndexError("Out of range")
     contador = 1
     current = self._head
     while contador <= i:
          contador += 1
          current = current._next
     return current._element
```

Implementando LDE

```
class LDE:
    def add(self, e, pos):
        n = Nodo(e, None, None)
        if self.empty:
            self._head = n
            self._tail = n
        elif pos > len(self)+1:
            raise IndexError("Out of range")
        elif pos == 0:
            n._next = self._head
            self._head._prev = n
            self._head = n
        elif pos == -1 or pos == len(self):
            n._prev = self._tail
            self._tail._next = n
            self._tail = n
        else:
            contador = 1
            current = self._head
            while contador < pos:
                current = current._next
                contador += 1
            n._prev = current
            n._next = current._next
            current._next._prev = n
            current._next = n

        self._len += 1
```

Arrays vs Listas enlazadas

Arrays	Listas Enlazadas
Permiten acceder rápidamente al n-ésimo elemento.	Se debe recorrer la lista hasta el n-ésimo elemento.
Agregar elementos en los extremos requiere solo ubicar la posición en memoria (operación aritmética).	Agregar elementos en los extremos requiere redirigir punteros.
Requieren solo memoria para los elementos de la lista.	Requieren memoria para los elementos de la lista y los punteros.
Es costoso mover todos los elementos para insertar/quitar uno del medio.	Una inserción/deleción en el medio de la lista solo requiere cambiar punteros.
Es posible necesitar mover todos los elementos al agregar/quitar elementos.	Una vez definido un nodo, no es necesario moverlo.

Bibliografía

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Capítulo 7
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press. Capítulo 10