Arrays y listas

...

Algoritmos y Estructuras de Datos
2024

UNRaf UNIVERSIDAD NACIONAL DE RAFAELA

Memoria de una computadora

- La memoria está compuesta por *bits* (0s y 1s), agrupados en unidades más grandes, como los *bytes* (8 bits)
- La computadora guarda donde está almacenado cada dato usando **direcciones de memoria**.
- Al saber donde está almacenado, puede acceder rápidamente a cada variable y las variables pueden modificar su contenido sin cambios grandes en el código.

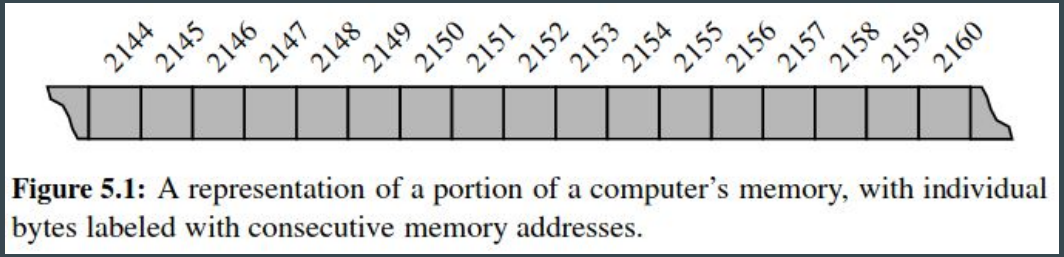


Figure 5.1: A representation of a portion of a computer's memory, with individual bytes labeled with consecutive memory addresses.

¿Cómo cree que están almacenados los datos en una lista?

Arrays

- Un grupo de variables relacionadas pueden guardarse de forma continua en la memoria. Estas estructuras se denominan ***arrays***.
- Un ejemplo son los ***strings***, que se pueden definir como una secuencia ordenada de caracteres individuales con un **tamaño fijo**.
- Todos los datos de un *array* deben ser del mismo tipo, para calcular fácilmente cuántos bytes son necesarios para almacenar todo un array y encontrar un bloque de memoria disponible de ese tamaño.

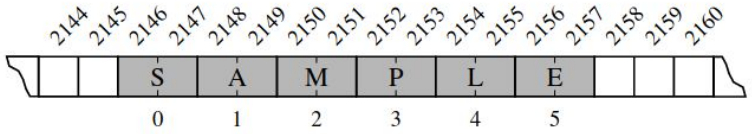


Figure 5.2: A Python string embedded as an array of characters in the computer's memory. We assume that each Unicode character of the string requires two bytes of memory. The numbers below the entries are indices into the string.

Arrays dinámicos

- Por definición, un *array* no puede cambiar de tamaño, ya que puede sobreescribir fragmentos de memoria que estén siendo usados.
- Para agregar elementos a un *array* se debe buscar un nuevo espacio en memoria que permita almacenar todos los datos, crear un nuevo *array* y eliminar el anterior.
- La clase ***list*** de Python es un ejemplo de arrays dinámicos.

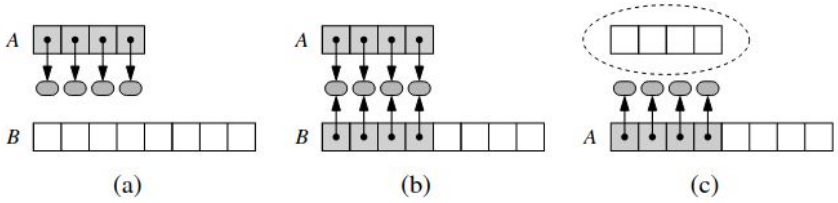


Figure 5.12: An illustration of the three steps for "growing" a dynamic array: (a) create new array *B*; (b) store elements of *A* in *B*; (c) reassign reference *A* to the new array. Not shown is the future garbage collection of the old array, or the insertion of the new element.

Arrays referenciados

- Es posible almacenar dentro de un array elementos de distintos tamaños.
- En lugar de guardar el elemento, se almacena la dirección de memoria donde se encuentra ese elemento.
- Todas las direcciones de memoria tienen el mismo tamaño, por lo que se cumple la condición necesaria que define un array.
- Una variable que almacena una dirección de memoria se la denomina **puntero**.

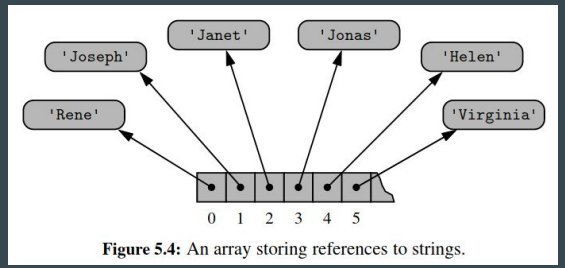


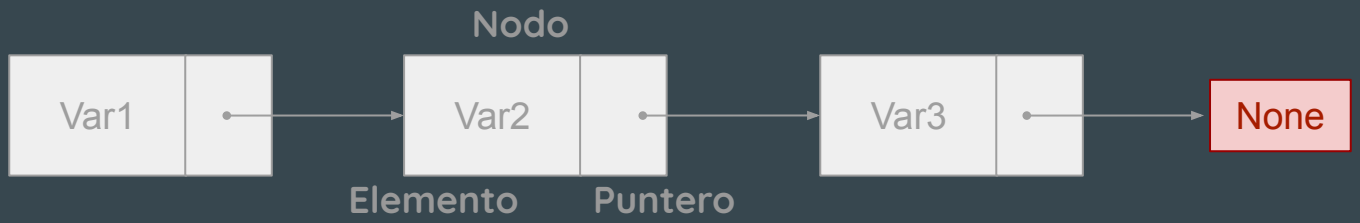
Figure 5.4: An array storing references to strings.

Listas de Python

- Las listas de Python son una clase sumamente optimizada y, usualmente, es la mejor opción para almacenar datos.
- Los datos son organizados como en un array.
- Sin embargo, posee algunas particularidades:
  - 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.

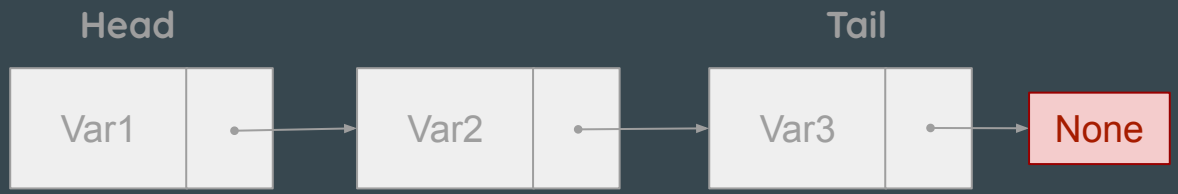
Listas enlazadas

- 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***).
- Siempre es necesario conocer el primer elemento de la lista; conocer el último no es indispensable, pero permite que algunas funciones sean más eficientes.



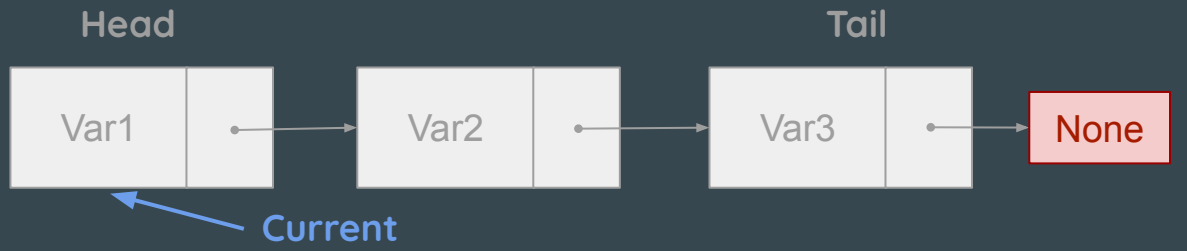
Implementando LSE

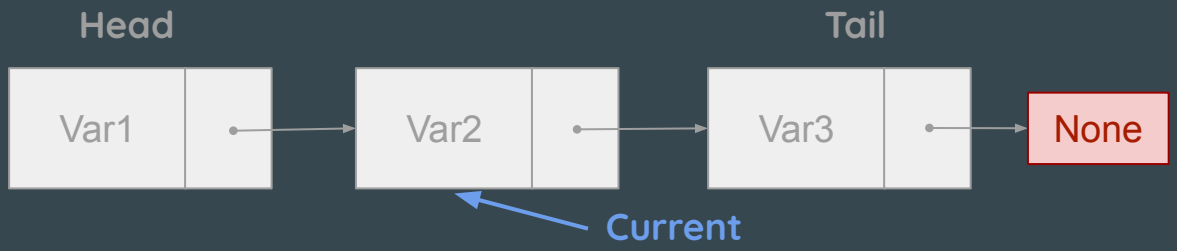
```
class NodeLSE:
  def __init__(self, e):
    self._element = e
    self._next = None

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
    else:
      return False
```

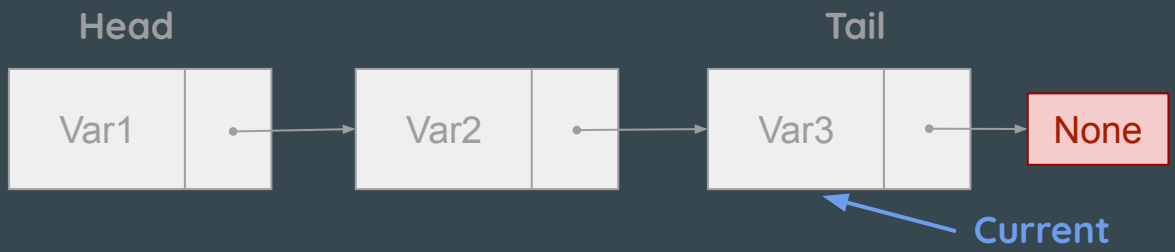
Recorrido de LSE

- 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





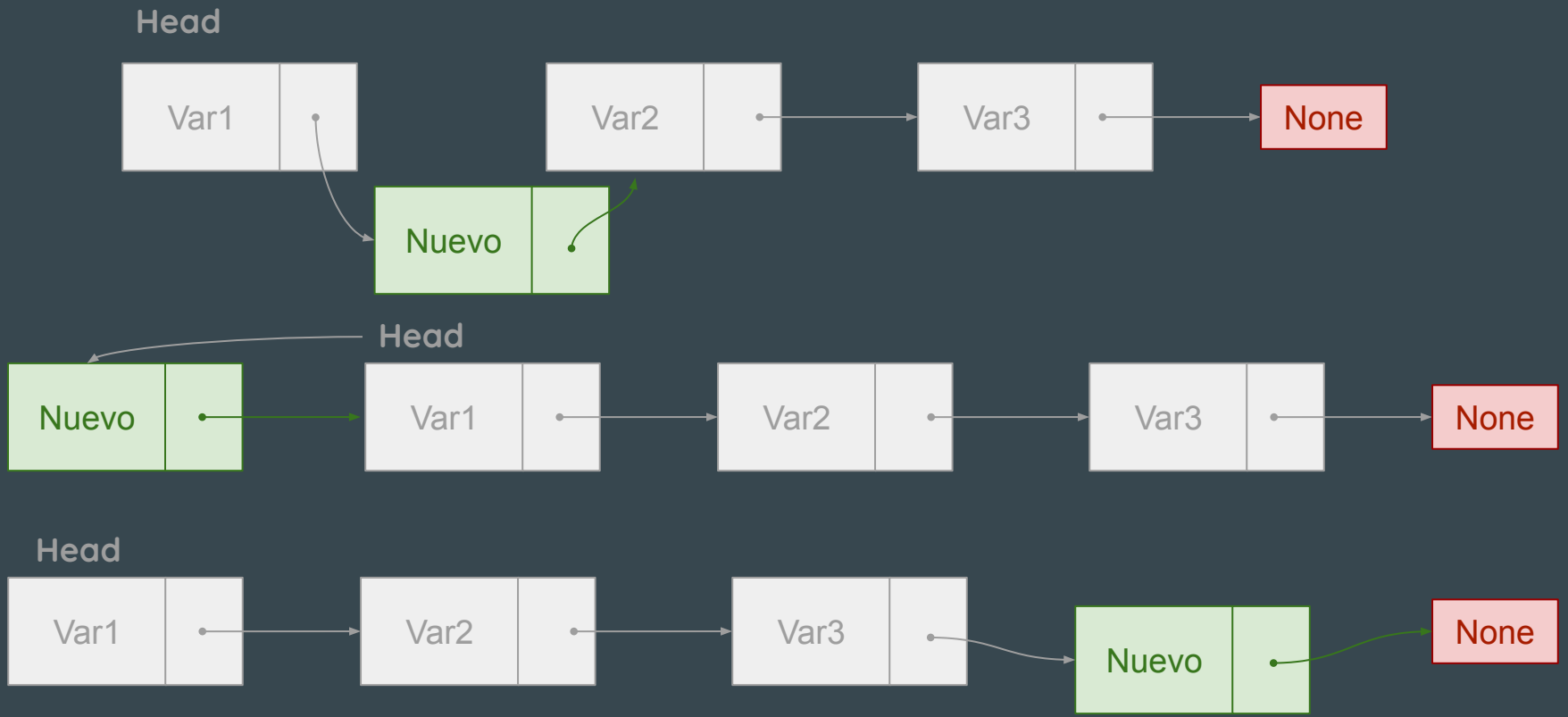



```
current = self._head
while current._next != None:
    current = current._next
```

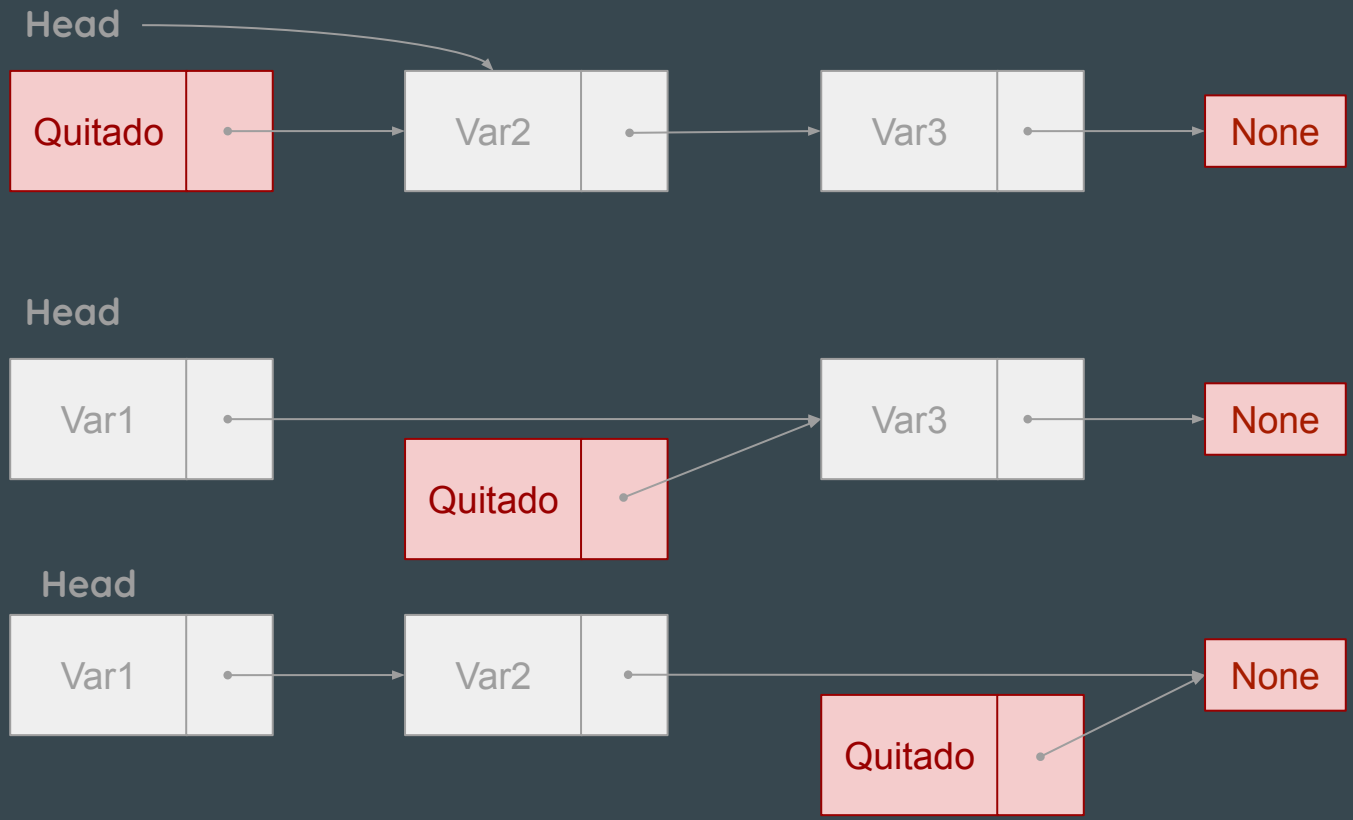
Recorriendo la LSE

```
class LSE:
  def print(self):
    current = self._head
    result = "["
    while current._next != None:
      result = result + str(current._element) + ","
      current = current._next
    result = result + str(current._element) + "]"
    print(result)

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

Añadir un elemento a la LSE



Quitar un elemento de la LSE



Agregando nodos de LSE

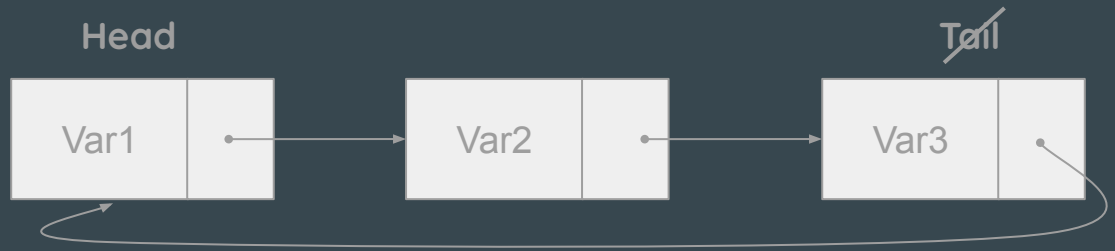
```
class LSE:
  def add_start(self, e):
    node = NodeLSE(e)
    if not self.empty():
      node._next = self._head
    self._head = node
    self._len += 1

  def add_end(self, e):
    node = NodeLSE(e)
    current = self._head
    while current._next != None:
      current = current._next
    current._next = node
    self._len += 1

  def add_in_position(self, e, p):
    # ???
```

Quitando nodos de LSE

```
class LSE:
  def remove_start(self):
    if self.empty():
      raise IndexError("List is empty")
    self._head = self._head._next
    self._len -= 1

  def remove_last(self):
    if self.empty():
      raise IndexError("List is empty")
    current = self._head
    prev = ""
    while current._next != None:
      prev = current
      current = current._next
    prev._next = None
    self._len -= 1

  def remove_from_position(self,p):
    # ???
```

Listas circulares (LC)

- Son similares a las LSE, pero el último elemento apunta al nodo ***head***.
- 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***

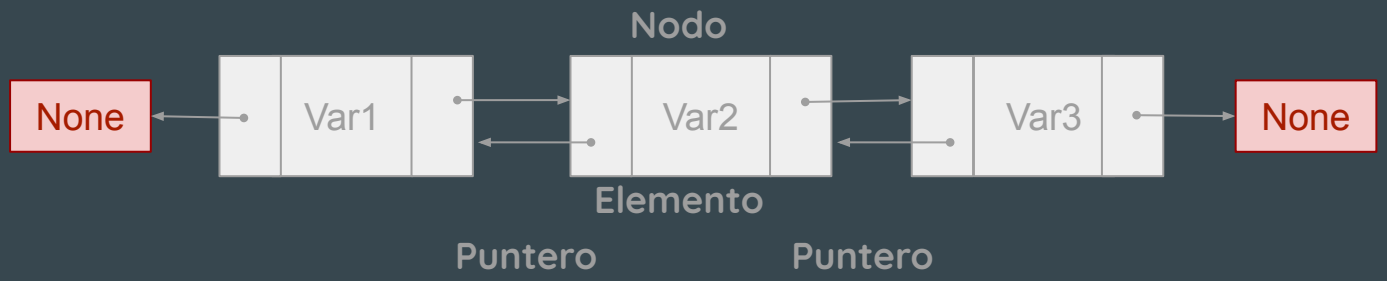


Trabajando con LC

- 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):
    self._element = e
    self._next = None
    self._prev = None

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

  #def remove_from_position?
  #def add_in_position?
  #???
```

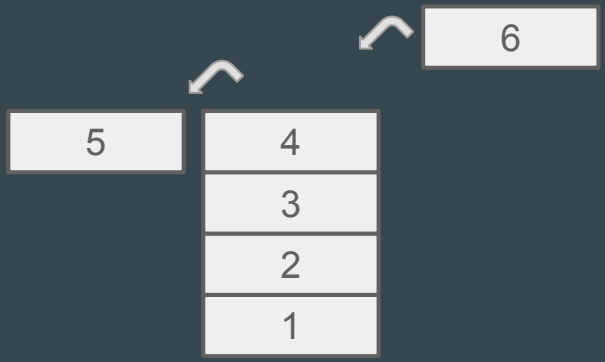
Arrays vs Listas enlazadas

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.

Casos especiales de listas

Pilas

- Las pilas son colecciones de objetos, similares a las listas, pero con reglas específicas para agregar y quitar nuevos objetos.
- Las pilas siguen el principio ***Last in, First out*** o **LIFO**.



Métodos en pilas

- empty: devuelve *True* si la pila no contiene ningún elemento
- push: agrega un elemento al tope de la pila.
- pop: elimina y devuelve el elemento del tope de la pila
- top: devuelve una referencia al elemento del tope de la pila sin eliminarlo
- len(S): devuelve el número de elementos de la pila

Ejemplo de funcionamiento de una pila

Operation	Return Value	Stack Contents
S.push(5)		
S.push(3)		
len(S)		
S.pop()		
S.empty()		
S.pop()		
S.empty()		
S.pop()		
S.push(7)		

Operation	Return Value	Stack Contents
S.push(9)		
S.top()		
S.push(4)		
len(S)		
S.pop()		
S.push(6)		
S.push(8)		
S.pop()		
S.top()		

Ejemplo de funcionamiento de una pila

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop()	3	[5]
S.empty()	False	[5]
S.pop()	5	[]
S.empty()	True	[]
S.pop()	Error	[]
S.push(7)	–	[7]

Operation	Return Value	Stack Contents
S.push(9)	–	[7, 9]
S.top()	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop()	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop()	8	[7, 9, 6]
S.top()	6	[7, 9, 6]

Codificando pilas usando arrays (listas de Python)

```
class ArrayStack:
  def __init__ (self):
    self._data = []

  def __len__ (self):
    return len(self._data)

  def empty(self):
    if len(self._data) == 0:
      return True
    else:
      return False


  def top(self):
    if self.empty():
      raise IndexError("Stack is empty")
    return self._data[-1]
```

```
def print(self):
  print(self._data)

def push(self, e):
  self._data.append(e)

def pop(self):
  if self.empty():
    raise IndexError("Stack is empty")
  return self._data.pop()
```

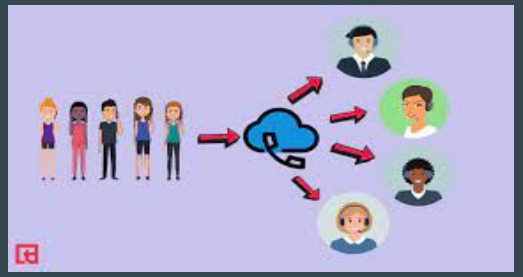
Codificando pilas usando listas enlazadas

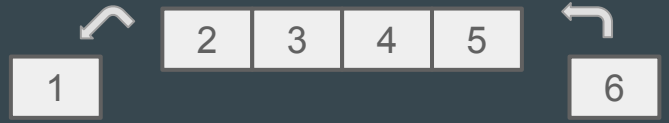
```
class NodeStack:
  def __init__(self, e):
    self._element = e
    self._prev = None

class ListStack:
  def __init__(self):
    self._top = None
    self._len = 0

  def __len__(self):
    return self._len

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

```
def top(self):
  if self.empty():
    raise IndexError("Stack is empty")
  return self._top._element

def push(self, e):
  node = NodeStack(e)
  if not self.empty():
    node._prev = self._top
  self._top = node
  self._len += 1

def pop(self):
  to_return = self._top._element
  self._top = self._top._prev
  self._len -= 1
  return to_return
```

Colas

- Otra estructura de datos fundamental son las ***colas***.
- En este caso, el funcionamiento sigue el principio ***First in, first out*** o ***FIFO***.





Métodos en colas

- empty: devuelve *True* si la cola no contiene ningún elemento
- enqueue: agrega un elemento al final de la cola.
- dequeue: elimina y devuelve el primer elemento de la cola
- first: devuelve una referencia al primer elemento de la cola sin eliminarlo
- len(S): devuelve el número de elementos de la cola

Ejemplo de funcionamiento de una cola

Operation	Return Value	Stack Contents
Q.enqueue(5)		
Q.enqueue(3)		
len(Q)		
Q.dequeue()		
Q.empty()		
Q.dequeue()		
Q.empty()		
Q.dequeue()		
Q.enqueue(7)		
Q.enqueue(9)		
Q.first()		
Q.enqueue(4)		
len(Q)		
Q.dequeue()		
Q.enqueue(6)		
Q.enqueue(8)		
Q.dequeue()		
Q.first()		

Ejemplo de funcionamiento de una cola

Operation	Return Value	Stack Contents
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue()	5	[3]
Q.empty()	False	[3]
Q.dequeue()	3	[]
Q.empty()	True	[]
Q.dequeue()	Error	[]
Q.enqueue(7)	–	[7]

Operation	Return Value	Stack Contents
Q.enqueue(9)	–	[7, 9]
Q.first()	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue()	7	[9, 4]
Q.enqueue(6)	–	[9, 4, 6]
Q.enqueue(8)	–	[9, 4, 6, 8]
Q.dequeue()	9	[4, 6, 8]
Q.first()	4	[4, 6, 8]

Codificando colas usando arrays (listas de Python)

```
class ArrayQueue:
  def __init__(self):
    self._data = []

  def __len__(self):
    return len(self._data)

  def empty(self):
    if len(self._data) == 0:
      return True
    else:
      return False

  def first(self):
    if self.empty():
      raise IndexError("Queue is empty")
    return self._data[0]
```

```
def print(self):
  print(self._data)

def enqueue(self, e):
    self._data.append(e)

def dequeue(self):
  if self.empty():
    raise IndexError("Queue is empty")
  return self._data.pop(0)
```

Codificando colas usando listas enlazadas

```
class NodeQueue:
  def __init__(self, e):
    self._element = e
    self._next = None

class ListQueue:
  def __init__(self):
    self._first = None
    self._len = 0

  def __len__(self):
    # Idem Pilas

  def empty(self):
    # Idem Pilas

def first(self):
  if self.empty():
    raise IndexError(“Queue is empty”)
  return self._first._element
```

```
def enqueue(self, e):
  node = NodeQueue(e)
  if not self.empty():
     current = self._first
     while current._next != None:
       current = current._next
     current._next = node
  else:
    self._first = node
  self._len += 1

def dequeue(self):
  if self.empty():
    raise IndexError("Queue is empty")
  to_return = self._first._element
  self._first = self._first._next
  self._len -= 1
  return to_return
```

Bibliografía

- Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data structures and algorithms in Python. Capítulo 6 y 7
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). Introduction to algorithms. MIT press. Capítulo 10