


5. Secuencias Enlazadas

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández

- ① Introducción
 - ② Secuencias enlazadas simples
 - ③ Pilas mediante secuencias enlazadas simples
 - ④ Colas mediante secuencias enlazadas simples
 - ⑤ Variantes de las secuencias
 - Secuencias circulares
 - Secuencias doblemente enlazadas
 - ⑥ Bibliografía
- 

Introducción

- Implementación del TAD Pila y Cola usando como base un array
- Desventajas:
 - La longitud de un array (dinámica) puede ser mayor que la cantidad real de elementos almacenados.
 - El tiempo requerido de ejecución de una secuencia de operaciones promediado sobre todas sus operaciones puede ser inaceptable en sistemas de tiempo real.
 - Las inserciones y eliminaciones en posiciones interiores de un array son costosas.

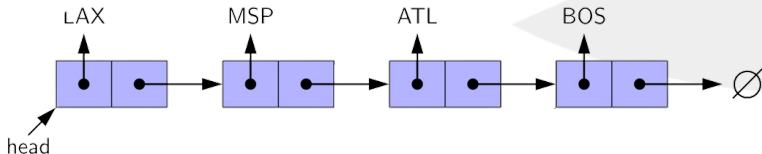
Secuencia enlazada: Alternativa a la list

- Mantienen los elementos en un orden determinado.
- Basada en array: representación centralizada...
 - Una porción de memoria tiene referencias a muchos elementos.
- Secuencia enlazada: representación distribuida...
 - Se asigna un objeto ligero (nodo) a cada elemento.
 - Cada nodo referencia a su elemento y a nodos vecinos para representar colectivamente el orden lineal de la secuencia.
- Desventajas:
 - No hay acceso eficiente a los elementos mediante índice numérico.
 - Examen de un nodo no es suficiente para saber su posición.

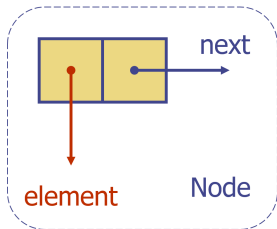
Secuencias enlazadas simples

Secuencia enlazada simple

- Estructura de datos formada por una secuencia de **nodos** que empieza en un nodo “head”.
 - Cada nodo (objeto único) almacena una referencia a un objeto (elemento de la secuencia) y una al siguiente nodo de la secuencia (el último None).
 - Otro objeto representa la secuencia; mantiene una referencia a la cabeza de la lista.
 - No es necesaria una referencia al final de la secuencia.
 - Requiere el recuento del número total de nodos (tamaño); evita recorrer la secuencia para contar.

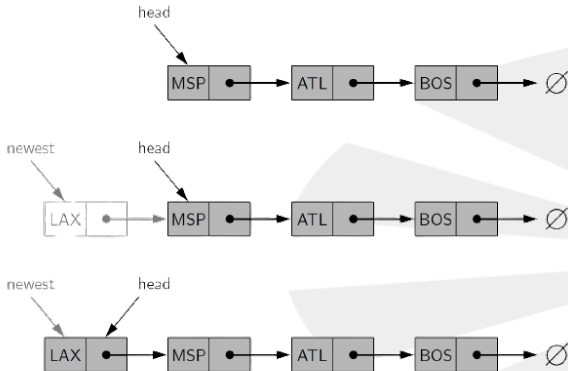


Definición de la clase Node



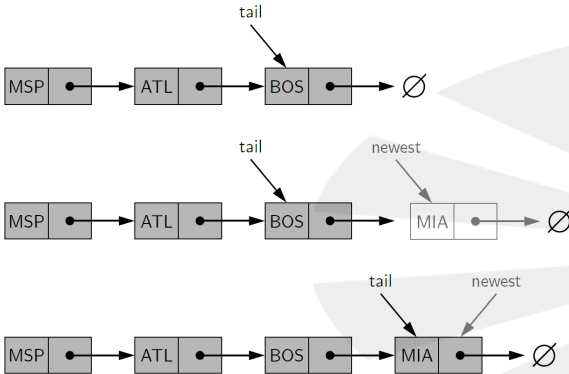
```
class Node:
    def __init__(self, element):
        self._element = element
        self._next = None
```


Insertar por el comienzo de la secuencia (no vacía)



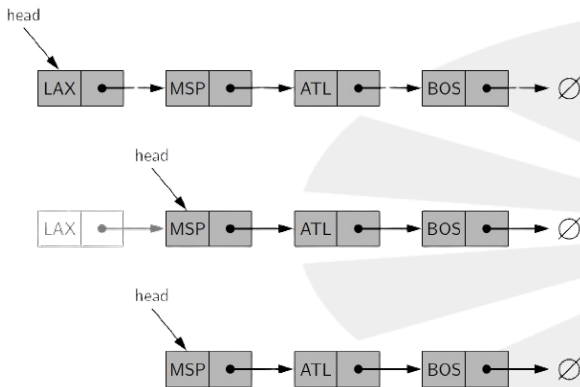
```
newest = Node(LAX)
newest._next = head
head = newest
```

Insertar por el final de la secuencia (no vacía)



```
newest = Node(MIA)
tail._next = newest
tail = newest
```

Eliminar por el comienzo de la secuencia



`head = head._next`

Eliminar por el final de la secuencia

- Si solo hay un elemento:

```
head = None
```

- Si hay más de un elemento, hay que recorrer la secuencia para encontrar el penúltimo y poder “desenganchar” el último:

```
temp = head
while temp._next._next is not None:
    temp = temp._next
temp._next = None
```

Sobre los nodos eliminados

- Cuando ya no hay ninguna referencia a un nodo, este deja de ser accesible.
- ¿Siguen ocupando memoria?
Sí ... hasta que el **Garbage collector** reclame el espacio que ocupan.
- Una parte del Garbage collector (Reference counting) está totalmente fuera del control del programador, otra parte (Generational Garbage collector) puede ser manejada mediante el módulo gc.
- En situaciones normales no deberíamos preocuparnos del Garbage collector.

Pilas mediante secuencias enlazadas simples

Pilas mediante secuencias enlazadas

```
class LinkedStack:
    class _Node:
        def __init__(self, element, next):
            self._element = element
            self._next = next

    def __init__(self):
        self._head = None
        self._size = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0
```

```
    def push(self, e):
        self._head = self._Node(e, self._head)
        self._size += 1

    def peek(self):
        if self.is_empty():
            raise Empty('Stack is empty')
        return self._head._element

    def pop(self):
        if self.is_empty():
            raise Empty('Stack is empty')
        answer = self._head._element
        self._head = self._head._next
        self._size -= 1
        return answer
```

Colas mediante secuencias enlazadas simples

Colas mediante secuencias enlazadas

```
class LinkedQueue:

    class _Node:
        def __init__(self, element, next):
            self._element = element
            self._next = next

    def __init__(self):
        self._head = None
        self._tail = None
        self._size = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0
```

Colas mediante secuencias enlazadas

Insertar por el final:

```
def first(self):
    if self.is_empty():
        raise Empty('Queue is empty')
    return self._head._element

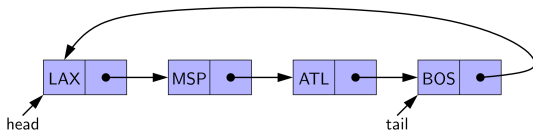
def enqueue(self, e):
    newest = self._Node(e, None)
    if self.is_empty():
        self._head = newest
    else:
        self._tail._next = newest
    self._tail = newest
    self._size += 1
```

Borrar por el comienzo:

```
def dequeue(self):
    if self.is_empty():
        raise Empty('Queue is empty')
    answer = self._head._element
    self._head = self._head._next
    self._size -= 1
    if self.is_empty():
        self._tail = None
    return answer
```

Variantes de las secuencias

Secuencias circulares



- Permite modelar “secuencias infinitas”, útil para técnicas *round-robin*¹.
- También resulta útil para modelar secuencias finitas. A tener en cuenta:
 - Cuidado con no “pasarse” del final de la lista.
 - Cuidado con no caer en ciclos.
 - Como *head* es ahora el nodo siguiente de *tail*, podemos prescindir del primero.
- Un ejemplo típico: la implementación de colas.

¹Dividir el tiempo de CPU de manera uniforme entre los procesos en ejecución, asignando a cada uno un pequeño intervalo.

Cola mediante secuencia circular

```
class CircularQueue:

    class _Node:
        def __init__(self, element, next):
            self._element = element
            self._next = next

    def __init__(self):
        self._tail = None
        self._size = 0

    def __len__(self):
        return self._size

    def is_empty(self):
        return self._size == 0
```

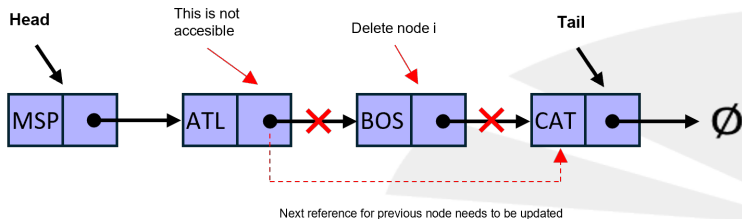
Cola mediante secuencia circular

```
def first(self):
    if self.is_empty():
        raise Empty('Queue is empty')
    head = self._tail._next
    return head._element

def enqueue(self, e):
    newest = self._Node(e, None)
    if self.is_empty():
        newest._next = newest
    else:
        newest._next = self._tail._next
        self._tail._next = newest
    self._tail = newest
    self._size += 1
```

```
def dequeue(self):
    if self.is_empty():
        raise Empty('Queue is empty')
    oldhead = self._tail._next
    if self._size == 1:
        self._tail = None
    else:
        self._tail._next = oldhead._next
    self._size -= 1
    return oldhead._element
```

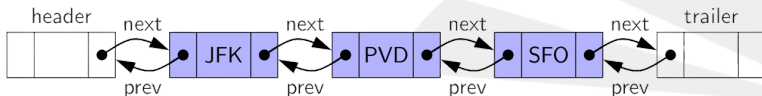
Limitaciones simplemente enlazadas



- Es difícil un borrado eficiente al final (tail) de la lista.
- No es posible un borrado eficiente de un nodo interior de la lista sólo con la referencia del nodo.
 - No se puede determinar el nodo que *precede* al que va a ser borrado.
 - Se necesita actualizar su referencia al siguiente.

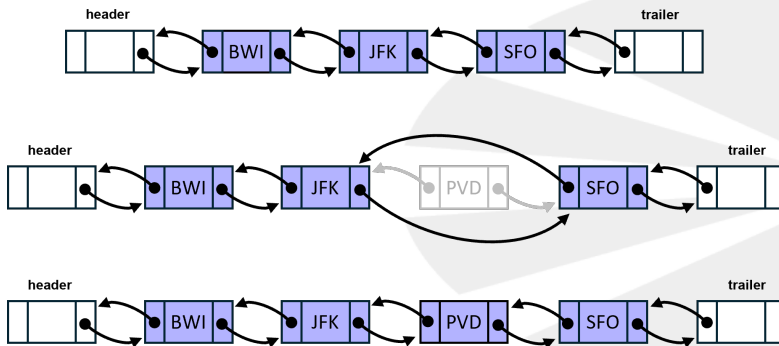
Secuencias doblemente enlazadas

```
class _Node:
    def __init__(self, element, prev, next):
        self._element = element
        self._prev = prev
        self._next = next
```

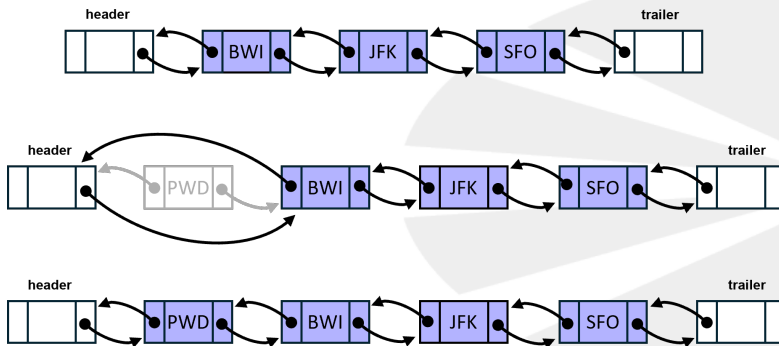


- Los nodos *header* y *trailer* (centinelas) no son imprescindibles.
- Con ellos inserción y borrado se realizan de la misma manera en cualquier lugar válido de la secuencia.
 - El nuevo nodo siempre estará situado entre un par de nodos existentes.
 - Cada elemento que se va a eliminar se almacena en un nodo que tiene vecinos a cada lado.

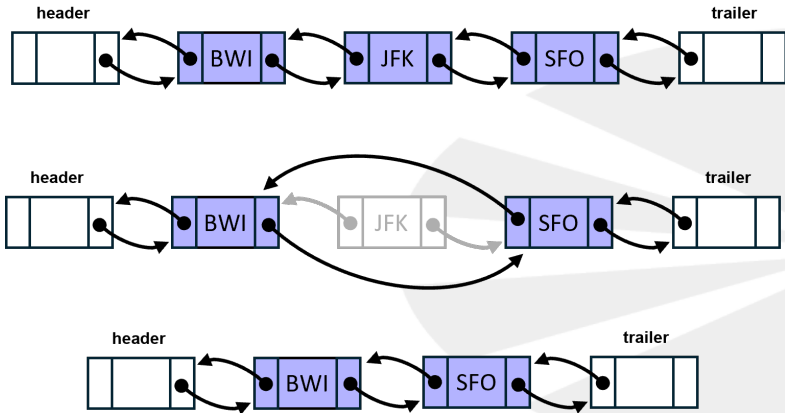
Inserción en secuencias doblemente enlazadas



Inserción en secuencias doblemente enlazadas



Borrado en secuencias doblemente enlazadas



- El uso de centinelas permite usar la misma implementación para borrar el primer o el último nodo.

Implementación: definición de nodos y constructor

```
class _DoublyLinkedBase:
    """A base class providing a doubly linked list representation."""

    #----- clase _Node anidada -----
    class _Node:
        def __init__(self, element, prev, next):
            self._element = element          # elemento almacenado
            self._prev = prev                 # referencia al nodo anterior
            self._next = next                 # referencia al nodo siguiente

    #----- constructor de la secuencia-----

    def __init__(self):
        self._header = self._Node(None, None, None) # crea una secuencia vacía
        self._trailer = self._Node(None, None, None) # crea nodo cabecera
        self._header._next = self._trailer           # crea nodo final
        self._trailer._prev = self._header           # enlaza los nodos...
        self._size = 0                               # ...cabecera y final
        # 0 elementos almacenados
```

Implementación: métodos públicos

```
def __len__(self):  
    return self._size  
  
def is_empty(self):  
    return self._size == 0
```

Implementación: métodos privados

```
# crea un nuevo nodo con el elemento "e" entre dos nodos existentes
def _insert_between(self, e, predecessor, successor):
    newest = self._Node(e, predecessor, successor)
    predecessor._next = newest
    successor._prev = newest
    self._size += 1
    return newest

# elimina un nodo de la secuencia y devuelve el elemento que contenía
def _delete_node(self, node):
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
    self._size -= 1
    element = node._element
    # Los nodos borrados tienen todos los campos a None
    # Será útil en las listas posicionales para validar "posiciones"
    node._prev = node._next = node._element = None
    return element
```

Bibliografía



- Capítulo 7 de: Michael T. Goodrich, Roberto Tamassia, and Michael H. Goldwasser. 2013. Data Structures and Algorithms in Python (1st edition). Wiley Publishing.
- Capítulo 4 de: Kenneth A. Lambert. 2018. Fundamentals of Python: Data Structures (2nd edition). Cengage.

5. Secuencias Enlazadas

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández