

Programación III

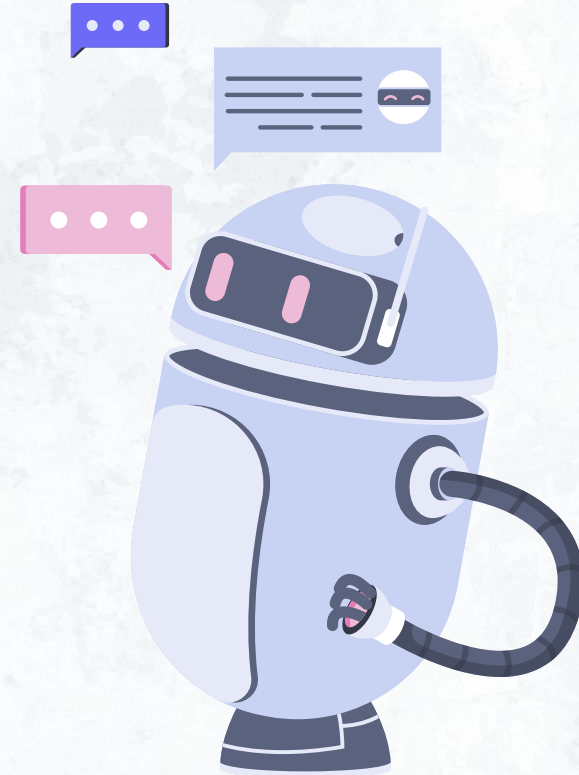
Implementación de Listas Enlazadas en Pilas y Colas - Tarea 02

MARIO LEANDRO CASTILLO SANHUEZA

UNIVERSIDAD CATÓLICA DE TEMUCO

Objetivos de la clase

- 1 → Pila con una Lista Enlazada Simple
- 2 → Cola con una Lista Enlazada Simple
- 3 → Tarea 02



Pila con una Lista Enlazada Simple

Al diseñar dicha implementación, debemos decidir si modelar la parte superior de la pila (top) en la cabeza o en la cola de la lista.

Hay una elección claramente mejor aquí: solo podemos insertar y eliminar elementos de manera eficiente en **tiempo constante** en la **cabeza**

Utilizaremos la clase Node ya creada para realizar esta implementación

```
class Node:
    """Node class for a singly linked list (optimized with __slots__)"""
    __slots__ = '_element', '_next' # Memory optimization

    def __init__(self, element, next_node=None):
        self._element = element # Data stored in the node
        self._next = next_node # Reference to the next node
```

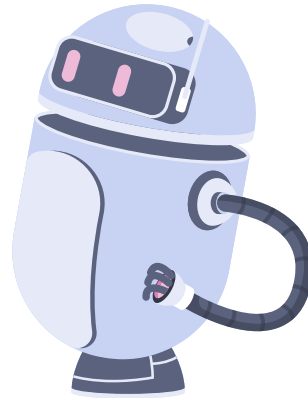
Pila con una Lista Enlazada Simple

La variable head es una referencia al nodo en la cabeza de la lista (o None si la pila está vacía). Llevamos el conteo del número actual de elementos con la variable de instancia size, ya que de lo contrario tendríamos que recorrer toda la lista para contar los elementos cuando se consulte el tamaño de la pila.

1. **La implementación de push es insertar en la cabeza de una lista enlazada simple.**
2. **Cuando implementamos el método top, el objetivo es retornar el elemento que está en el tope de la pila.**
3. **En la implementación del pop guardamos una referencia local al elemento que está siendo eliminado y lo retornamos al llamador de pop.**

Pila con una Lista Enlazada Simple

- Como todas las operaciones se hacen en el "tope", es más eficiente poner el tope en la cabeza de la lista.
- Se define una clase interna (`_Node`) que representa un nodo con dos partes:
 - El elemento que guarda.
 - El siguiente nodo de la lista.
- La pila tiene dos atributos:
 - `_head`: el nodo en la cima.
 - `_size`: cuántos elementos hay en total.



Pila con una Lista Enlazada Simple

Con Lista

Operation	Running Time
<code>S.push(e)</code>	$O(1)$
<code>S.pop()</code>	$O(1)$
<code>S.top()</code>	$O(1)$
<code>len(S)</code>	$O(1)$
<code>S.is_empty()</code>	$O(1)$

Sin Lista

Operation	Running Time
<code>S.push(e)</code>	$O(1)^*$
<code>S.pop()</code>	$O(1)^*$
<code>S.top()</code>	$O(1)$
<code>S.is_empty()</code>	$O(1)$
<code>len(S)</code>	$O(1)$

*amortized

Pila con una Lista Enlazada Simple

```
class LinkedStack:
    """LIFO Stack implementation using a singly linked list for storage."""
    # ----- Nested Node class -----
    class _Node:
        """Lightweight, non-public class for storing a singly linked node."""
        __slots__ = '_element', '_next'


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

Clase Nodo anidada




Pila con una Lista Enlazada Simple

```
# ----- Stack methods -----  
  
def __init__(self):  
    """Create an empty stack."""  
    self._head = None # reference to the head  
    self._size = 0     # number of elements  
  
def __len__(self):  
    """Return the number of elements in the stack"""  
    return self._size  
  
def is_empty(self):  
    """Return True if the stack is empty."""  
    return self._size == 0
```



El constructor mantiene la cabeza (frente de la pila) y el tamaño real



Los métodos para obtener el largo y si la pila está vacía ahora utilizarán el dato size de la clase

Pila con una Lista Enlazada Simple

```
def push(self, e):  
    """Add element e to the top of the stack."""  
    self._head = self._Node(e, self._head)  
    self._size += 1
```

Push ahora establecerá la cabeza de la pila como el dato tope por lo que este cambia regularmente.

```
def top(self):  
    """Return (but do not remove) the element at the top of the stack.  
  
    Raise OwnEmpty exception if the stack is empty.  
    """  
    if self.is_empty():  
        raise OwnEmpty("The stack is empty")  
    return self._head._element # top of stack is at the head
```

Top ahora utilizará head como el puntero para obtener el último elemento

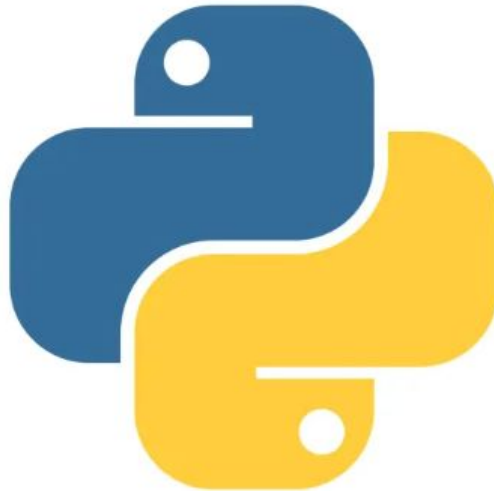
Pila con una Lista Enlazada Simple

```
def pop(self):  
    """Remove and return the element from the top of the stack (LIFO).  
  
    Raise OwnEmpty exception if the stack is empty.  
    """  
    if self.is_empty():  
        raise OwnEmpty("The stack is empty")  
    answer = self._head._element  
    self._head = self._head._next # bypass the forward pointer  
    self._size -= 1  
    return answer
```

El pop guardará el dato que es la cabecera actual para retornarlo y luego realizará el intercambio de puntero

Pila con una Lista Enlazada Simple

Código completo : `Linked_Stack`



Cola con una Lista Enlazada Simple



Así como hicimos con el TDA de pila, podemos usar una lista enlazada simple para implementar el TDA de cola mientras se mantiene un tiempo de ejecución en el peor de los casos de $O(1)$ para todas las operaciones.

Como necesitamos realizar operaciones en ambos extremos de la cola, mantendremos explícitamente una referencia tanto al inicio (head) como al final (tail) como variables de instancia para cada cola.

La orientación natural para una cola es alinear el frente de la cola con el inicio de la lista, y la parte posterior de la cola con el final de la lista, ya que debemos poder encolar (enqueue) elementos al final, y desencolar (dequeue) desde el frente.

Cola con una Lista Enlazada Simple



Debemos definir una clase **LinkedQueue** con una clase interna **Node**, igual que en **LinkedStack**.

A diferencia de la pila, se almacenan dos referencias:

- head: al principio de la lista (el frente de la cola).
- tail: al final de la lista (el fondo de la cola).

Esto permite operaciones eficientes tanto en el frente como en el fondo.

Cola con una Lista Enlazada Simple



1. enqueue(e):

- Inserta un nuevo nodo al final (tail) de la lista.
- Si la cola estaba vacía, el nuevo nodo se vuelve tanto el head como el tail.
- En otro caso, se conecta como siguiente del tail actual y se actualiza tail.

1. dequeue():

- Elimina el nodo al inicio (head) de la lista y retorna su valor.
- Si después de eliminar el nodo, la cola queda vacía, se actualiza también tail = None.

1. first():

- Retorna el elemento en el frente de la cola, sin eliminarlo.
- Lanza una excepción si la cola está vacía.

1. is_empty() y __len__():

- Verifican si la cola está vacía o retornan la cantidad de elementos.

Cola con una Lista Enlazada Simple

```
class LinkedQueue:
    """FIFO queue implementation using a singly linked list for storage."""

    class Node:
        """Lightweight, non-public class for storing a single element"""
        __slots__ = 'element', 'next'

        def __init__(self, element, next_node):
            self.element = element
            self.next = next_node

    def __init__(self):
        """Create an empty queue."""
        self.head = None
        self.tail = None
        self.size = 0 # number of elements in the queue
```

Clase Nodo anidada

El constructor posee el head, tail y el tamaño real.

Cola con una Lista Enlazada Simple

```
def __len__(self):  
    """Return the number of elements in the queue."""  
    return self.size
```

← Len retorna el size real

```
def is_empty(self):  
    """Return True if the queue is empty."""  
    return self.size == 0
```


← Empty trabaja con el size real

```
def first(self):  
    """Return (but do not remove) the element at the front of the queue.  
  
    Raise OwnEmpty if the queue is empty.  
    """  
    if self.is_empty():  
        raise OwnEmpty("Queue is empty")  
    return self.head.element # front aligned with head of list
```

← El método first utiliza el puntero al head de la cola

Cola con una Lista Enlazada Simple

```
def dequeue(self):  
    """Remove and return the first element of the queue (FIFO).  
  
    Raise OwnEmpty if the queue is empty.  
    """  
    if self.is_empty():  
        raise OwnEmpty("Queue is empty")  
    answer = self.head.element  
    self.head = self.head.next  
    self.size -= 1  
    if self.is_empty(): # special case: queue is now empty  
        self.tail = None # the removed node was also the tail  
    return answer
```



Al desencolar guardamos el elemento que esté en la cabeza para retornarlo y luego intercambiamos el puntero al dato siguiente.

Cola con una Lista Enlazada Simple

```
def enqueue(self, e):  
    """Add an element to the back of the queue."""  
    newest = self.Node(e, None) # node will become new tail  
    if self.is_empty():  
        self.head = newest # special case: queue was empty  
    else:  
        self.tail.next = newest  
    self.tail = newest # update reference to tail  
    self.size += 1
```

Al encolar ahora verificamos si la cola está o no vacía para asignar este dato como la cabeza si se cumple

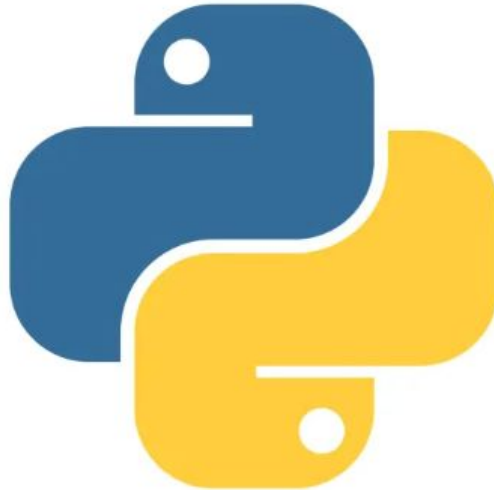
Si ya existe una cabeza entonces tenemos que agregar este dato al final de la cola, cambiar el puntero y sumar uno al tamaño real.

Cola con una Lista Enlazada Simple



UNIVERSIDAD
CATÓLICA DE
TEMUCO

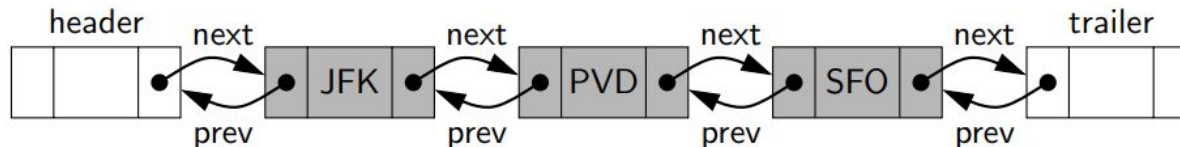
Código completo : `Linked_Queue`



Lista Doblemente-enlazada en la vida real

(Caso aeropuerto)

- ¿Donde se aplica este tipo de lista?
 - Estructura eficiente para gestionar secuencias de vuelos (despegues, aterrizajes, prioridades).
 - Permite ajustar dinámicamente el orden de operaciones en pista, abordando cambios en tiempo real.
- Ejemplo típico:
 - **Priorización de vuelos:**
 - Emergencias (fallas técnicas, combustible, etc) se insertan al frente de la cola para atención inmediata.
 - Vuelos regulares (programados) se agregan al final.
 - **Gestionar retrasos:**
 - Reordenar vuelos afectados por demoras hacia posiciones intermedias sin alterar toda la secuencia.
- ¿Por qué usar una cola doblemente-enlazada?
 - Adaptabilidad: Insertar o eliminar vuelos urgentes sin romper el flujo de operaciones programadas.
 - Rapidez: Operaciones en $O(1)$ para añadir/eliminar en ambos extremos.



ACTIVIDAD - CONTEXTO

Imagina que trabajas en una torre de control de un aeropuerto concurrido: cada minuto llegan nuevos vuelos, algunos con emergencias médicas, otros con retrasos por clima, y todos compitiendo por un espacio limitado en las pistas. Los controladores aéreos necesitan:

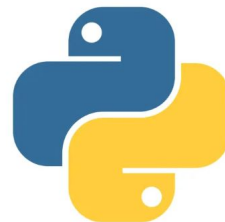
1. Priorizar vuelos de emergencia (que deben saltar al frente de la lista)
2. Mantener el orden básico de los vuelos programados
3. Organizar dinámicamente cuando hay retrasos o cancelaciones
4. Consultar rápidamente qué vuelos están próximos a despegar

Desafío: Las estructuras de datos simples como arrays o colas no son suficientes, un array tendría costos altos al reordenar y una cola solo permite acceso a los extremos, por lo tanto necesitamos algo más flexible...

Objetivo general : Crear una lista Doblemente-enlazada para la Gestión de Vuelos con ORM (SQLAlchemy)

Objetivos de Aprendizaje:

- Modelar un sistema de gestión de vuelos en aeropuertos usando una Lista doblemente-enlazada en un contexto real.
- Integrar estructuras de datos con persistencia (base de datos).
- Desarrollar una API REST (FastAPI) para operaciones CRUD + undo/redo.



ACTIVIDAD - Requisitos Técnicos

1. Estructura de Datos Obligatoria (TDA Lista Doblemente-enlazada)

Debe incluir como mínimo: `insertar_al_frente(vuelo)`: # Añade un vuelo al inicio de la lista (para emergencias).

`insertar_al_final(vuelo)`: # Añade un vuelo al final de la lista (vuelos regulares).

`obtener_primer()`: # Retorna (sin remover) el primer vuelo de la lista.

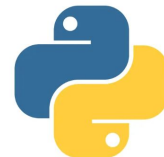
`obtener_ultimo()`: # Retorna (sin remover) el último vuelo de la lista

`longitud()`: # Retorna el número total de vuelos en la lista

`insertar_en_posicion(vuelo, posicion)`: # Inserta un vuelo en una posición específica (ej: índice 2).

`extraer_de_posicion(posicion)`: # Remueve y retorna el vuelo en la posición dada (ej: cancelación).

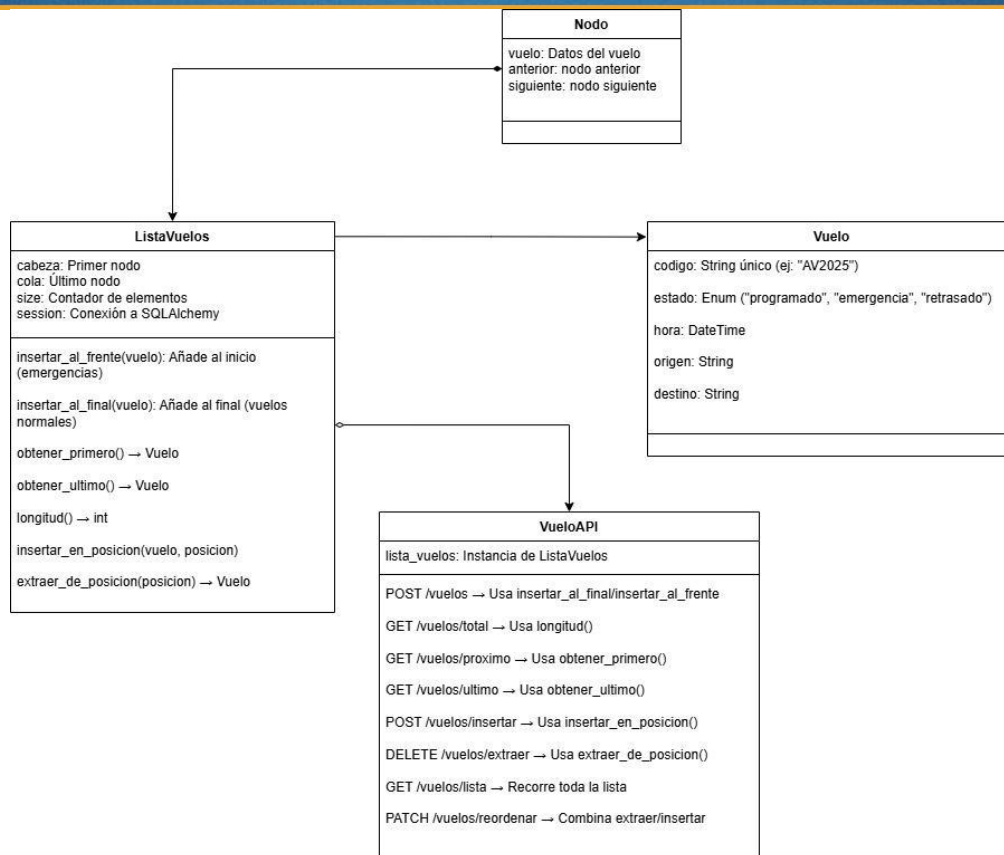
*Puedes renombrar los métodos a tu elección pero todos deben estar presentes y en uso.



2. Persistir los vuelos en una base de datos con relaciones básicas (tabla Vuelos), a través de los siguientes endpoint requeridos. (FastAPI)

MÉTODO	ruta	DESCRIPCIÓN
POST	/VUELOS	AÑADE UN VUELO AL FINAL (NORMAL) O AL FRENTE (EMERGENCIA).
GET	/VUELOS/TOTAL	RETORNA EL NÚMERO TOTAL DE VUELOS EN COLA.
GET	/VUELOS/PROXIMO	RETORNA EL PRIMER VUELO SIN REMOVER.
GET	/VUELOS/ULTIMO	RETORNA EL ÚLTIMO VUELO SIN REMOVER.
POST	/VUELOS/INSERTAR	INSERTA UN VUELO EN UNA POSICIÓN ESPECÍFICA.
DELETE	/VUELOS/EXTRAER	REMUEVE UN VUELO DE UNA POSICIÓN DADA.
GET	/VUELOS/LISTA	LISTA TODOS LOS VUELOS EN ORDEN ACTUAL.
PATCH	/VUELOS/REORDENAR	REORDENA MANUALMENTE LA COLA (EJ: POR RETRASOS).

ACTIVIDAD - DIAGRAMA DE APOYO



Documentacion: <https://docs.sqlalchemy.org/en/20/orm/quickstart.html>

Que debe entregar:

- Video explicativo del trabajo realizado, compartido mediante un enlace de Youtube. (**Duración máximo 10 min**).
 - **Explicar donde se utiliza la Lista doblemente-enlazada**
 - **Explicar donde se utiliza el ORM.**
 - **Explicar donde se utiliza la API.**
- Código subido a un repositorio en Github.
- Trabajo Individual.
- Plazo de entrega: 22/04





UNIVERSIDAD
CATÓLICA DE
TEMUCO

¿CONSULTAS?

MARIO LEANDRO CASTILLO SANHUEZA

UNIVERSIDAD CATÓLICA DE TEMUCO