**FUNDAMENTOS DE PROGRAMACIÓN**

# Proyecto Entregable: Entrega 02

La evaluación continua requiere de la realización de un proyecto a lo largo de todo el cuatrimestre que cuenta con un total de 3 entregas, cada una de ellas con una puntuación. El objetivo de la segunda práctica es implementar y trabajar con diferentes tipos agregados en Python, enfocándose en estructuras de datos que son fundamentales para la gestión y organización de elementos. En concreto se trabajará con las siguientes estructuras: agregado lineal, lista ordenada, lista ordenada sin repetición, cola, cola de prioridad y pila.

## 1. Construye el proyecto

Usando el mismo proyecto que utilizó para la primera entrega, siga los siguientes pasos:

1. Cree el paquete entrega2.tipos y dentro de él cree los módulos Agregado_lineal.py, Lista_ordenada.py, Lista_ordenada_sin_repeticion.py, Cola.py, Cola_prioridad.py, Pila.py.
2. Para poder probar todos los tipos cree un paquete test dentro de entrega2 y cree un test por cada módulo anteriormente creado.

Si ha seguido las instrucciones correctamente su estructura de proyecto debería verse de la siguiente forma:


Recuerde que las siguientes importaciones son necesarias para la creación de clases (al margen de otras que pudiera necesitar):

```
from __future__ import annotations
```

**IMPORTANTE**: Para la creación de estos tipos **NO** se puede usar el decorador *@dataclass*

## A. Agregado lineal <E>

Un agregado lineal es una colección que almacena elementos en el orden en que se agregan, permitiendo operaciones de inserción y eliminación. Para este tipo se diseñará una clase abstracta que luego será implementada mediante varias subclases.

**Propiedades**

- **elements**, de tipo List[E] protegida: Esta propiedad será una lista que almacenará los elementos añadidos.
- ***size(self) -> int,*** derivada: Indica el número de elementos en el agregado.
- ***is_empty(self) -> bool***, derivada: Devuelve True si el agregado no tiene elementos y False en caso contrario.
- ***elements(self) -> list[E]***, derivada***:*** Devuelve la lista completa de elementos almacenados en el agregado.

**Otros métodos**

- ***add(self, e: E) -> None (abstracto):*** Este método será abstracto y deberá ser implementado por las subclases. Será el responsable de añadir un nuevo elemento al agregado.
- ***add_all(self, ls: list[E]) -> None:*** Añade una lista de elementos al agregado, utilizando el método add.
- ***remove(self) -> E:*** Elimina y devuelve el primer elemento agregado. Antes de realizar la eliminación, el método debe verificar que la lista no está vacía utilizando una sentencia *assert len(self._elements) > 0, 'El agregado está vacío'*. Si la lista está vacía, la ejecución debe detenerse mostrando este mensaje de error.
- ***remove_all(self) -> list[E]:*** Elimina todos los elementos, devolviendo una lista con los elementos eliminados. Tenga en cuenta que deberá implementar este método de tal forma que reutilice los métodos *is_empty()* y *remove().*

## B. Lista_ordenada <Agregado_lineal[E], Generic[E,R]>

El tipo *Lista_ordenada* hereda de la clase abstracta *Agregado_lineal.* Esta lista debe ordenar automáticamente los elementos según un criterio de orden que se especificará en la creación del objeto.

**Propiedades**

- **order,** de tipo Callable[[E], R], privada. Indica el criterio de ordenación por el que van a estar ordenados los elementos.

**Métodos de factoría**

- ***of(order: Callable[[E], R]) -> Lista_ordenada[E, R]:*** Construye un objeto a partir de una función que especifica el criterio de orden que se utilizará para comparar los elementos. A continuación se dejan algunos criterios de orden a modo de ejemplo:
  - lista = Lista_ordenada.of(lambda x: x)
  - lista_str = Lista_ordenada.of(lambda x: len(x))

**Otros métodos**

- ***__index_order(self, e: E) -> int:*** Este método privado calcula el índice en el que se debe insertar el nuevo elemento, de acuerdo con la función de ordenación. Si el elemento es menor que todos los elementos presentes, se inserta al principio (posición 0); si es mayor que todos, se inserta al final (longitud de la lista); En caso de que *e* se encuentre entre otros elementos, devuelve el índice donde se puede insertar *e* para mantener el orden.
- ***add(self, e: E) -> None:*** Añade un nuevo elemento a la lista, insertándolo en la posición correcta para mantener el orden. Tenga en cuenta que para que este método funcione correctamente debe reutilizar *__index_order*.

**Representación como cadena**:
ListaOrdenada($e_1$, $e_2$, $e_3$,…, $e_n$)

## C. Lista_ordenada_sin_repeticion <Agregado_lineal[E], Generic[E,R]>

Este tipo es una implementación de una estructura de datos que extiende la clase *Agregado_lineal.* Permite almacenar elementos de forma ordenada sin permitir duplicados.

**Propiedades**

- **order,** de tipo Callable[[E], R], privada. Indica el criterio de ordenación por el que van a estar ordenados los elementos.

**Métodos de factoría**

- ***of(order: Callable[[E], R]) -> Lista_ordenada_sin_repeticion[E, R]:*** Construye un objeto a partir de una función que especifica el criterio de orden que se utilizará para comparar los elementos.

**Otros métodos**

- ***__index_order(self, e: E) -> int:*** Este método privado calcula el índice en el que se debe insertar el nuevo elemento, de acuerdo con la función de ordenación. Si el elemento es menor que todos los elementos presentes, se inserta al principio (posición 0); si es mayor que todos, se inserta al final (longitud de la lista); En caso de que *e* se encuentre entre otros elementos, devuelve el índice donde se puede insertar *e* para mantener el orden.
- ***add(self, e: E) -> None:*** Añade el elemento e a la lista solo si este no está ya presente. En el caso de que no esté presente se deberá insertar en la posición correcta para mantener el orden. Tenga en cuenta que para que este método funcione correctamente debe reutilizar *__index_order*.

**Representación como cadena**:
ListaOrdenadaSinRepeticion($e_1$, $e_2$, $e_3$,…, $e_n$)

## D. Cola <Agregado_lineal[E]>

Este tipo es una implementación de una estructura de datos de tipo *FIFO* (First In, First Out) que extiende la clase *Agregado_lineal*. Esta clase permite almacenar elementos de manera secuencial, donde los elementos se añaden al final de la cola y se eliminan desde el principio. Es ideal para gestionar tareas en orden de llegada, como en la programación de procesos o la gestión de recursos.

**Métodos de factoría**

- ***of()-> Cola[E]:*** Método de factoría que crea e inicializa una nueva instancia de la clase.

**Otros métodos**

- ***add(self, e: E) -> None:*** Añade el elemento *e* al final de la cola.

**Representación como cadena**:
Cola($e_1$, $e_2$, $e_3$,…, $e_n$)

## E. Cola de prioridad <Generic[E,P]>

Este tipo representa una estructura de datos que gestiona elementos junto con sus prioridades. Extiende la funcionalidad básica de una cola al permitir que cada elemento tenga una prioridad asociada, determinando el orden de atención en función de esta prioridad. Los elementos con mayor prioridad son atendidos primero, lo que permite una gestión eficiente de tareas o procesos que requieren atención en función de su urgencia. A continuación, se expone un ejemplo ilustrativo:

Considera un hospital donde los pacientes son atendidos según la gravedad de su condición. Supongamos que llegan tres pacientes:

- **Paciente A**: Dolor de cabeza leve (prioridad 3)
- **Paciente B**: Fractura en la pierna (prioridad 2)
- **Paciente C**: Ataque cardíaco (prioridad 1)

En la cola de prioridad, se añadirán de la siguiente manera:

1. Paciente A (3)
2. Paciente B (2)
3. Paciente C (1)

La cola se organizará así:

[(‘Paciente A’, 3), (‘Paciente B’, 2), (‘Paciente C’, 1)

**Propiedades**

- **elements**, de tipo List[E] protegida: Esta propiedad almacena los elementos de la cola de prioridad en el orden determinado por sus prioridades.
- **priorities,** de tipo List[P] protegida: almacena las prioridades asociadas a cada elemento en elements, permitiendo determinar el orden de atención.
- ***size(self) -> int,*** derivada: Indica el número de elementos en el agregado.
- ***is_empty(self) -> bool***, derivada: Devuelve True si el agregado no tiene elementos y False en caso contrario.
- ***elements(self) -> list[E]***, derivada***:*** Devuelve la lista completa de elementos almacenados en el agregado.

**Otros métodos**

- ***add(e: E, priority: P) -> None:*** añade un elemento e con una prioridad priority a la cola. La posición de inserción se determina en función de la prioridad, manteniendo el orden correcto.
- ***add_all(self, ls: list[tuple[E,P]) -> None:*** Añade una lista de elementos al agregado, utilizando el método add.

- ***remove(self) -> E:*** Elimina y devuelve el elemento con la mayor prioridad (el primero en la lista). Antes de realizar la eliminación, el método debe verificar que la lista no está vacía utilizando una sentencia *assert len(self._elements) > 0, 'El agregado está vacío'*. Si la lista está vacía, la ejecución debe detenerse mostrando este mensaje de error.
- ***remove_all(self) -> list[E]:*** Elimina todos los elementos, devolviendo una lista con los elementos eliminados. Tenga en cuenta que deberá implementar este método de tal forma que reutilice los métodos *is_empty()* y *remove().*
- ***index_order(self, priority: P) -> int:*** Este método privado calcula el índice en el que se debe insertar el nuevo elemento, de acuerdo con su prioridad.
- ***decrease_priority(self, e:E, new_priority:P) -> None:*** Disminuye la prioridad de un elemento *e*. Si la nueva prioridad es menor que la anterior, se elimina el elemento y se vuelve a añadir con la nueva prioridad, manteniendo el orden.

**Representación como cadena**:
ColaPrioridad[($e_1$,$p_1$), ($e_2$, $p_2$), …, ($e_n$, $p_n$) )]

## F. Pila <Agregado_lineal[E]>

Una Pila es una estructura de datos que sigue el principio Last In, First Out (LIFO), lo que significa que el último elemento agregado es el primero en ser retirado. En una pila, los elementos se apilan uno encima de otro, y solo se puede acceder al elemento que está en la parte superior de la pila. Esta estructura es útil en diversas aplicaciones, como el seguimiento de la historia de navegación en un navegador web, la gestión de llamadas en programación, y la resolución de problemas que requieren retroceso (backtracking).

Tenga en cuenta que *Pila* hereda del tipo *Agregado Lineal.*

**Métodos de factoría**

- ***of()-> Pila[E]:*** Método de factoría que crea e inicializa una nueva instancia de la clase.

**Otros métodos**

- ***add(e: E) -> None:*** añade un elemento e a la parte superior de la pila (al principio).

**ANEXO**

A continuación, se muestra a modo de guía una posible salida por consola de cada uno de los tests a realizar.

## *Test_Lista_ordenada.py*

```
TEST DE LISTA ORDENADA:


################################################
Creación de una lista con criterio de orden lambda x: x
Se añade en este orden: 3, 1, 2
Resultado de la lista: ListaOrdenada([1, 2, 3])


################################################
El elemento eliminado al utilizar remove(): 1


################################################
Elementos eliminados utilizando remove_all: [1, 2, 3]


################################################
Comprobando si se añaden los números en la posición correcta...
Lista después de añadirle el 0: ListaOrdenada([0, 1, 2, 3])
Lista después de añadirle el 10: ListaOrdenada([0, 1, 2, 3, 10])
Lista después de añadirle el 7: ListaOrdenada([0, 1, 2, 3, 7, 10])
```

## *Test_Lista_ordenada_sin_repeticion.py*

```
TEST DE LISTA ORDENADA SIN REPETICIÓN:


################################################
Creación de una lista con criterio de orden lambda x: -x
Se añade en este orden: 23, 47, 47, 1, 2, -3, 4, 5
Resultado de la lista ordenada sin repetición: ListaOrdenadaSinRepeticion([47, 23, 5, 4, 2, 1, -3])


################################################
```

```
El elemento eliminado al utilizar remove(): 47


################################################

Elementos eliminados utilizando remove_all: [47, 23, 5, 4, 2, 1, -3]


################################################

Comprobando si se añaden los números en la posición correcta...

Lista después de añadirle el 0: ListaOrdenadaSinRepeticion([47, 23, 5, 4, 2, 1, 0, -3])

Lista después de añadirle el 0: ListaOrdenadaSinRepeticion([47, 23, 5, 4, 2, 1, 0, -3])

Lista después de añadirle el 7: ListaOrdenadaSinRepeticion([47, 23, 7, 5, 4, 2, 1, 0, -3])
```

## *Test_Cola.py*

```
TEST DE COLA:


################################################

Creación de una cola vacía a la que luego se le añaden con un solo método los números: 23, 47, 1, 2, -3, 4, 5

Resultado de la cola: Cola([23, 47, 1, 2, -3, 4, 5])


################################################

Elementos eliminados utilizando remove_all: [23, 47, 1, 2, -3, 4, 5]
```

## *Test_Cola_prioridad.py*

Para este test se da el código que debe utilizar y ejecutar, si la implementación ha sido correcta la salida de dicho test debe ser: **Pruebas superadas exitosamente.**

```
from entrega2.tipos.Cola_prioridad import *


def test_cola_prioridad():
    cola = Cola_de_prioridad[str, int]()

    # Agregar pacientes
    cola.add('Paciente A', 3)  # Dolor de cabeza leve
    cola.add('Paciente B', 2)  # Fractura en la pierna
    cola.add('Paciente C', 1)  # Ataque cardíaco

    # Verificar el estado de la cola
    assert cola.elements() == ['Paciente C', 'Paciente B', 'Paciente A'], "El orden de la cola es incorrecto."

    # Atender a los pacientes y verificar el orden de atención
```

```
    atencion = []
    while not cola.is_empty():
        atencion.append(cola.remove())

    assert atencion == ['Paciente C', 'Paciente B', 'Paciente A'], "El orden de atención no es correcto."

    print("Pruebas superadas exitosamente.")

if __name__ == '__main__':
    test_cola_prioridad()
```