Facultad de Ingeniería
UNER

# Trabajo Práctico N° 2

ALUMNOS:

Biasoli, Ana Inés

Taborda, Verónica

# Problema 1

El triaje es un proceso clave en contextos médicos donde la cantidad de pacientes supera la capacidad de atención inmediata. Su objetivo es clasificar a los pacientes según la gravedad de su estado de salud, priorizando la atención de los más críticos.

En esta simulación, los pacientes llegan al centro de salud con uno de los siguientes niveles de riesgo:

- **1** → **Crítico**
- **2** → **Moderado**
- **3** → **Bajo**

Actualmente, los pacientes son atendidos en el orden de llegada. Esto no refleja una atención médica eficiente, ya que no prioriza a quienes están en situación más delicada.

Objetivos:

Implementar una **estructura de datos genérica** (es decir, no específica para pacientes) que permita:

- Insertar pacientes a medida que llegan.
- Atender siempre al paciente con **mayor prioridad médica** (riesgo más bajo numéricamente).
- En caso de empate en el nivel de riesgo, **priorizar al que haya llegado antes**.

Para cumplir con los requerimientos, se utiliza una **cola de prioridad genérica,** capaz de almacenar cualquier tipo de dato. Está basada en un **montículo binario mínimo (heap-min) propio**, que permite acceder al elemento de **menor valor** de forma eficiente (en este caso, el paciente con menor número de riesgo, o sea el más urgente). Este montículo no utiliza módulos estándar como heapq.

Cada elemento ingresa a la cola se representa por una tupla formada por 3 elementos:

```
(prioridad, orden_de_llegada, objeto)
```

siendo:

- prioridad el nivel de riesgo
- orden_de_llegada el contador incremental que va a determinar en caso de empate quien se atiende primero, es decir quien tiene mayor prioridad;
- objeto es un dato, que en este caso es el paciente.

El montículo que implementamos va a mantener ordenados los elementos según la prioridad, garantizando que al primero que se atienda sea el de mayor urgencia médica.

## Complejidad

- **Inserción (`push`)**: O(log n); lo que hace es que agrega un elemento al final del arreglo y lo reordena mediante `heapify_up`

- **Eliminación (`pop`)**: O(log n); este extrae el primer elemento (mínimo) y reorganiza mediante `heapify_down`.

## Ventajas

Las ventajas de esta solución son:

- La estructura permite atender primero a los pacientes con mayor riesgo, ya que el orden se da en base a este.

- en caso de que haya dos pacientes con el mismo riego, el desempate se da por el orden de llegada

- **Separación clara** entre la estructura genérica (`colaprioridadQueue`, `mont_binario`) y el uso concreto (`paciente`).

- El rendimiento de la estructura  se mantiene eficiente incluso con una gran cantidad de pacientes.

# Problema 2

Análisis de complejidad O grande para cada método

| Método | Función | Complejidad temporal | Justificación |
|---|---|---|---|
| guardar_temperatura (fecha, temp) | Inserta o actualiza una medición de temperatura para una fecha | O(log n) | Insertar o buscar una fecha en un AVL es logarítmico |
| devolver_temperatura (fecha) | Devuelve la temperatura de una fecha exacta | O(log n) | Búsqueda en árbol AVL |
| temperatura_maxima _rango(f1, f2) | Devuelve la temperatura máxima dentro de un rango de fechas | O(k + log n) | Búsqueda y recorrido parcial en un rango; k es la cantidad de fechas dentro del rango |
| temperatura_minima_ rango(f1, f2) | Igual al anterior, pero para la temperatura mínima | O(k + log n) | Igual al caso anterior |
| temperaturas_extrem os_rango(f1, f2) | Devuelve temperaturas mínimas y máximas dentro de un rango | O(k + log n) | Recorre todos los nodos en rango |
| devolver_temperatura s(f1, f2) | Lista todas las temperaturas en un rango de fechas, ordenadas | O(k + log n) | Búsqueda y recorrido ordenado entre f1 y f2 |
| borrar_temperatura(fe cha) | Elimina una medición por fecha | O(log n) | Eliminación en AVL |
| cantidad_muestras() | Devuelve cuántas muestras hay almacenadas | O(1) | Se mantiene un contador de nodos |
| __str__() (o impresión del árbol) | Devuelve todas las temperaturas ordenadas por fecha | O(n) | Requiere recorrer todos los nodos en orden |

Para probar el correcto funcionamiento de los métodos de **temperatura_db** implementamos un test en el cual se importó el la clase `Temperaturas_DB`, se define una clase llamada pruebas en la cual se cargan datos de ejemplo para poder corroborar el correcto funcionamiento de las funciones definidas.

```python
from modules.temperatura_db import Temperaturas_DB
def pruebas():
    db = Temperaturas_DB()
    db.guardar_temperatura(22.5, "01/06/2024")
    db.guardar_temperatura(25.0, "02/06/2024")
    db.guardar_temperatura(20.3, "03/06/2024")
    db.guardar_temperatura(19.8, "04/06/2024")
    db.guardar_temperatura(21.2, "05/06/2024")
    db.guardar_temperatura(23.7, "06/06/2024")
    db.guardar_temperatura(18.9, "07/06/2024")
    db.guardar_temperatura(24.4, "08/06/2024")
    db.guardar_temperatura(26.0, "09/06/2024")
    db.guardar_temperatura(22.0, "10/06/2024")
```

Las líneas de la siguiente imagen verifican el correcto funcionamiento de las funciones, como guardar; valores máximos y mínimos en un rango, devolver las temperaturas en un rango, eliminar, y excepción en caso de un error de carga de datos.

```
15    print("Temperatura el 02/06:", db.devolver_temperatura("02/06/2024"))
16    print("Cambio de temperatura del 02/06 a 27.3:", db.guardar_temperatura(27.3, "02/06/2024"))
17    print("Temperatura máxima entre 01 y 04:", db.temperatura_maxima_rango("01/06/2024", "10/06/2024"))
18    print("Temperatura mínima entre 01 y 04:", db.temperatura_minima_rango("01/06/2024", "10/06/2024"))
19    print("Todas las muestras:", db.devolver_temperaturas_rango("01/06/2024", "10/06/2024"))
20    print("Temperattura del 03/06:", db.devolver_temperatura("03/06/2024"))
21    print("Elimino temperatura del 03/06:", db.eliminar_temperatura("03/06/2024"))
22    print("Temperaturas después de eliminar el 03/06:", db.devolver_temperaturas_rango("01/06/2024", "10/06/2024"))
23    print("Cantidad de muestras luego de eliminar:", db.cantidad_muestras())
24    print("Datos extremos entre 01/06 y 06/06:", db.extremos_temperatura_rango("01/06/2024", "06/06/2024"))
25    print("Rango sin datos del 10 al 12/06:", db.devolver_temperaturas_rango("11/06/2024", "15/06/2024"))
26    print("Temperatura del 05/06 antes de cambio:", db.devolver_temperatura("05/06/2024"))
27    print("Cambio de temperatura del 05/06 a calor:", db.guardar_temperatura("calor", "05/06/2024"))
28    print("Temperatura del 05/06 después de cambio:", db.devolver_temperatura("05/06/2024"))
29
30
31    print("\n ✓ Todas las pruebas pasaron.")
```

# Problema 3- Palomas Mensajeras

En este problema se busca determinar la forma más eficiente de propagar una noticia desde la aldea "Peligros" hacia otras 21 aldeas, utilizando una red de palomas mensajeras entrenadas. Cada aldea tiene conexión solo con ciertas aldeas vecinas, y se conoce la distancia entre ellas.

La consigna establece que:

- Cada aldea debe recibir la noticia una sola vez.
- Una vez que una aldea recibe la noticia, puede retransmitirla a todas sus vecinas.

El mensaje debe comenzar desde “Peligros” y recorrer la red minimizando el costo total de envío, es decir, la suma de todas las distancias recorridas por las palomas. Entonces, planteamos un árbol de propagación óptimo que conecte todas las aldeas usando la menor cantidad total de leguas recorridas, sin ciclos y asegurando que cada aldea esté conectada a la red.

Para resolver el problema se modela la red de aldeas como un grafo no dirigido y ponderado, donde:

- Cada aldea es un vértice.
- Cada conexión es una arista con un peso igual a la distancia entre aldeas.

Se implementa el algoritmo de Prim, que permite construir un Árbol de Expansión Mínima (MST) desde la aldea "Peligros". Este algoritmo garantiza que:

- La noticia llegue a todas las aldeas.
- La distancia total recorrida sea la mínima posible.
- No haya repetición de mensajes

Entrada
El programa lee las conexiones desde un archivo aldeas.txt, donde cada línea tiene el formato: [AldeaOrigen,AldeaDestino,Distancia]
Salida

- Se imprime la lista de aldeas en orden alfabético.
- Para cada aldea, se muestra:
    - Quién le envió la noticia (su predecesor),
    - A quiénes se la reenvía (siguientes).
- Finalmente, se informa la distancia total recorrida por las palomas.

Estructura modular

- `grafo.py`: estructura del grafo.
- `vertice.py`: definición de cada aldea.
- `cola_prioridad.py` y `monticulo_binario.py`: implementación eficiente de la cola de prioridad.
- `prim.py`: algoritmo de Prim adaptado.
- `main.py`: carga de datos, ejecución y salida de resultados.