Facultad de Ingeniería

# Algoritmos y Estructuras de datos

**Informe Trabajo Práctico N° 2**

**Integrantes: Obaid Aixa, Caraballo Mateo**

# Actividad 1

En nuestra simulación de atención en una sala de emergencias, reemplazamos la lista simple que se usaba originalmente por una cola de prioridad. De esta forma, logramos que siempre se atienda primero al paciente con el mayor nivel de riesgo (es decir, los casos críticos), sin importar el orden en que llegaron.
Para implementarlo, usamos un heap (montículo) con el módulo heapq de Python, que nos permite mantener la prioridad de forma eficiente. Además, agregamos un contador interno para resolver los casos en los que dos pacientes tienen el mismo nivel de riesgo: en esos casos, se atiende primero al que llegó antes. Esto respeta el criterio de triaje habitual, donde la urgencia médica es lo principal, pero también se tiene en cuenta el orden de ingreso cuando hay igualdad de condiciones.
La estructura que desarrollamos es genérica, por lo que no está pensada exclusivamente para pacientes. Puede almacenar cualquier tipo de dato, lo que la hace reutilizable en otros contextos.
Con esta solución, logramos que tanto la inserción como la eliminación de pacientes se hagan en tiempo logarítmico, y consultar al próximo en ser atendido lleva tiempo constante. Esto hace que el sistema funcione de forma rápida y ordenada, incluso si hay muchos pacientes esperando.

# Actividad 2

Implementamos la clase Temperaturas_DB usando un árbol AVL para almacenar y gestionar las temperaturas asociadas a fechas. Esta estructura permite mantener el árbol balanceado, lo que asegura eficiencia en las operaciones principales de inserción, búsqueda y eliminación.
El sistema permite guardar temperaturas, consultarlas por fecha o rango, obtener máximos y mínimos en un rango, borrar registros y consultar la cantidad de muestras almacenadas. Para probar el correcto funcionamiento, se desarrollaron casos de prueba que verifican cada operación fundamental, asegurando que la estructura se mantenga balanceada y las consultas respondan correctamente.

Tabla de Complejidades Big-O

| **Método** | **Complejidad** | **Explicación breve** |
|---|---|---|
| guardar_temperatura | O(log n) | Inserción en árbol AVL balanceado. |
| devolver_temperatura | O(log n) | Búsqueda en árbol AVL. |
| borrar_temperatura | O(log n) | Eliminación y rebalanceo en AVL. |
| max_temp_rango | O(k + log n) | Búsqueda en rango (k elementos en rango). |
| min_temp_rango | O(k + log n) | Similar a max_temp_rango. |
| temp_extremos_rango | O(k + log n) | Recorrido para mínimos y máximos en rango. |
| devolver_temperaturas | O(k + log n) | Devuelve listado ordenado de temperaturas. |
| cantidad_muestras | O(1) | Simple retorno del contador. |

Donde n es la cantidad total de nodos y k la cantidad de nodos dentro del rango consultado.

# Actividad 3

En esta actividad teníamos que encontrar la forma de repartir un mensaje a todas las aldeas sin pasar dos veces por la misma. La condición era que cada aldea solo podía enviar mensajes a sus aldeas vecinas. Para resolver esto, usamos el algoritmo de Prim, que nos permite armar un árbol de expansión mínima (o AEM), es decir, una estructura que conecta todos los nodos con el menor costo posible.

La cátedra nos compartió un archivo con los nombres de las aldeas y sus conexiones vecinas, y a partir de eso armamos nuestro grafo. Para la implementación, reutilizamos una clase que ya teníamos (MonticuloBinario) y desarrollamos dos clases nuevas: Vertice y Grafo, que son claves para poder usar Prim.

La clase Vertice representa cada aldea, guarda con qué otras aldeas está conectada, el peso (o distancia) de esas conexiones, y también la distancia mínima que tiene hasta el momento, junto con el vértice desde el que se llegó (su predecesor). Toda esta info es necesaria para que Prim pueda ir eligiendo siempre la mejor opción.

La clase Grafo, por su parte, se encarga de organizar todos los vértices y aristas. Tiene métodos para agregar vértices, crear aristas entre ellos y también para cargar todos los datos directamente desde un archivo, lo cual nos facilitó bastante el trabajo.

Prim empieza desde un nodo que elegimos como inicial (en nuestro caso, la aldea "Peligros"). A todos los vértices les asigna una distancia infinita al principio, menos al de inicio que arranca con 0. Después, usando el MonticuloBinario, va armando el árbol seleccionando siempre el vértice con menor distancia y actualizando las conexiones de sus vecinos. Esto hace que podamos repartir el mensaje a todas las aldeas de la forma más eficiente posible, sin repetir caminos ni perder tiempo.