UNER

FACULTAD DE INGENIERÍA

BIOINGENIERÍA

Algoritmos y Estructuras de datos

TP n°2

Aplicaciones de estructuras jerárquicas y grafos

GRUPO:7

INTEGRANTES:

-Rios, Rodrigo Ezequiel

-Schönfeldt, Andres

COMISIÓN:2

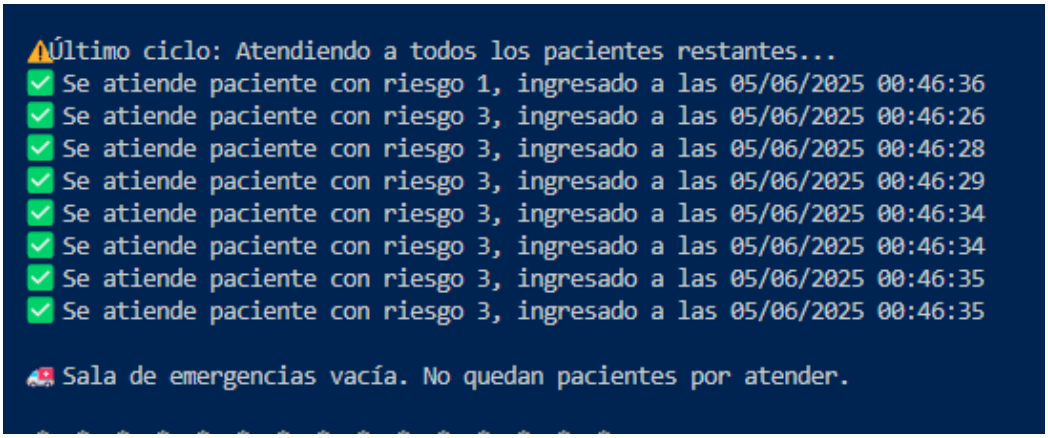
Problema 1

-Fundamentación de la estructura seleccionada

La estructura seleccionada para la gestión de pacientes en la sala de emergencias es una cola de prioridad basada en un montículo binario (heap). Esto permite atender primero a los pacientes con mayor nivel de riesgo, optimizando la selección mediante un criterio eficiente. La inserción de un nuevo paciente en la cola tiene una complejidad de O(log n), ya que el heap debe reorganizarse para mantener la prioridad. De manera similar, la eliminación del paciente más crítico también tiene un costo de O(log n), asegurando que la operación sea rápida incluso con un alto número de pacientes en espera. Comparado con una lista ordenada, que tendría inserciones O(n) y eliminaciones O(1), la estructura de heap mantiene un equilibrio adecuado entre rendimiento y acceso prioritario. Además, permite acceder al paciente con mayor riesgo en O(1), lo que mejora la eficiencia de selección sin necesidad de recorrer toda la lista. Esta implementación es crucial en escenarios donde la toma de decisiones debe hacerse en tiempo real, respetando criterios médicos de urgencia y minimizando tiempos de espera innecesarios.

Conclusión

El problema se resolvió mediante una **cola de prioridad basada en un montículo binario (heap)**, asegurando que los pacientes con mayor riesgo fueran atendidos primero. Se modularizó la solución en `paciente.py`, que define el objeto paciente con atributos de riesgo y orden de llegada, `cola_prioridad.py`, que implementa la estructura eficiente con `heapq`, y `main.py`, que gestiona la simulación. Durante 20 ciclos, varios pacientes ingresan por turno, acumulándose en la cola de espera. En cada ciclo, se atienden de uno a dos pacientes según prioridad. Al final del último ciclo, todos los pacientes han sido atendidos y la sala queda vacía.



Pseudocódigo del proceso:

```
inicializar_estructura()
mientras hay_pacientes():
```

```
    ingresar_pacientes_random()
    atender_pacientes_prioritarios()
mostrar_resultado_final()
```

La simulación usa aleatoriedad para generar pacientes y administrar su atención, manteniendo un flujo dinámico. La estructura del heap garantiza eficiencia en cada operación, respetando la prioridad médica.

Problema 2

El código consiste en una base de datos diseñada para gestionar un historial de mediciones de temperatura, asociando cada temperatura con una fecha específica. Internamente, esta clase utiliza un **Árbol AVL** (Árbol Binario de Búsqueda Auto-Balanceado) para almacenar los datos. Consta de las siguientes funciones:

-`guardar_temperatura()`: Añade o actualiza una temperatura para una fecha.

-`devolver_temperatura()`: Busca y retorna la temperatura de una fecha específica.

-`min_temp_rango()` y `max_temp_rango()`: Encuentran la temperatura mínima y máxima, respectivamente, dentro de un rango de fechas.

-`temp_extremos_rango()`: Retorna tanto la temperatura mínima como la máxima en un rango.

-`borrar_temperatura()`: Elimina un registro de temperatura por fecha.

-`devolver_temperaturas()`: Obtiene un listado formateado de todas las temperaturas dentro de un rango dado, ordenadas por fecha.

-`cantidad_muestras()`: Devuelve el número total de registros.

Salida de un ejemplo de uso del código:

```
Devolviendo temperatura para 05/01/2023:
28.0
Devolviendo temperatura para 06/01/2023:
None

Temperatura mínima en el rango 01/01/2023 a 10/01/2023:
22.1

Temperatura máxima en el rango 01/01/2023 a 10/01/2023:
30.2

Temperaturas extremas en el rango 01/01/2023 a 10/01/2023:
(22.1, 30.2)

Listado de temperaturas en el rango 01/01/2023 a 15/01/2023:
01/01/2023: 25.5°C
02/01/2023: 30.2°C
03/01/2023: 26.8°C
05/01/2023: 28.0°C
07/01/2023: 27.5°C
10/01/2023: 22.1°C
15/01/2023: 24.0°C

Borrando temperatura para 05/01/2023...
Temperatura para la fecha 05/01/2023 eliminada exitosamente.
Cantidad de muestras después de borrar: 7
Verificando si 05/01/2023 existe:
None

Intentando borrar una fecha que no existe (06/01/2023):
Advertencia: No se encontró temperatura para la fecha 06/01/2023.

Actualizando temperatura para 01/01/2023...
Temperatura para 01/01/2023: 26.0
Cantidad de muestras (no debería cambiar): 7
```

Tabla de análisis de complejidad de cada método:

Metodo	Complejidad	Explicación breve
Guardar	O(logn)	Los árboles AVL son estructuras de datos auto-balanceadas, lo que significa que mantienen su altura proporcional a logn. Por lo tanto, el tiempo necesario para encontrar la posición de inserción y realizar las rotaciones necesarias para mantener el balance es logarítmico.
Devolver Temperatura	O(logn)	Dado que el árbol AVL está siempre balanceado, la ruta desde la raíz hasta cualquier nodo (o hasta donde debería estar un nodo si no existe) nunca excede logn pasos.
Min Temp Rango	O(n)	Para encontrar los nodos, en el peor escenario (cuando el rango abarca casi todo el árbol, por ejemplo, de la primera a la última fecha registrada), la función tendrá que visitar un número de nodos proporcional a n. Aunque en el caso promedio puede ser O(k) donde k es el número de elementos en el rango.
Max Temp Rango	O(n)	Si el rango de fechas es muy amplio y abarca la mayoría de los nodos del árbol, la operación de recolectar esos nodos y luego encontrar el máximo en la lista resultante tomará un tiempo proporcional a la cantidad total de elementos, n.
Temp Extremo Rango	O(n)	En el peor escenario, donde el rango de fechas es extenso y se visitan muchos nodos, el costo de este recorrido y las operaciones subsiguientes en la lista serán proporcionales al número total de elementos (n).
Borrar	O(logn)	El método borrar busca el nodo a eliminar (lo cual toma O(logn)), y luego, si el nodo tiene dos hijos, encuentra su sucesor en orden (otra operación O(logn)). Finalmente, se realizan las rotaciones necesarias para reestablecer el balance del árbol, lo cual también se completa en tiempo logarítmico.
Devolver Temperaturas	O(n) o O(klogk)	El método utiliza el recorrido en orden (_recorrido_en_orden) para visitar los nodos dentro del rango especificado, lo cual es eficiente. Luego, se realiza una operación de ordenamiento (.sort()) sobre la lista de resultados. Aunque el recorrido en orden ya provee los elementos por fecha, la especificación de ordenar "por fechas" y la construcción de la lista formateada hacen que en el peor caso (cuando el rango cubre la mayoría de los elementos), la complejidad esté dominada por la creación de la lista y su posible ordenamiento explícito, llegando a O(n).
Cantidad Muestras	O(1)	Acceder al valor de una variable es una operación directa y no depende del número de elementos en el árbol, por lo que su tiempo de ejecución es constante.

Conclusión:

El uso de un Árbol AVL es crucial porque garantiza que todas las operaciones básicas (inserción, búsqueda, eliminación) mantengan una eficiencia de **O(logn)**, lo que es fundamental para bases de datos con un gran volumen de datos, ya que evita que las operaciones se vuelvan lentas a medida que se añaden más registros.
El código provee una solución eficiente y robusta para la gestión de temperaturas por fecha, aprovechando las ventajas de rendimiento de los

árboles AVL para las operaciones de inserción, búsqueda y eliminación de elementos individuales, aunque las operaciones de rango que requieren inspeccionar múltiples elementos pueden escalar linealmente con el tamaño del rango.

Problema 3

Conclusión

La solución al problema de **Palomas Mensajeras** se estructuró en módulos bien definidos, optimizando la comunicación entre aldeas mediante un **árbol de expansión mínima (MST)** con el algoritmo de **Prim**. `aldeas_data.py` gestionó la información de rutas, `graph.py` implementó la lógica del grafo, y `palomas_mensajeras_app.py` integró análisis y visualización, asegurando que cada aldea reciba la noticia solo una vez con el menor costo posible. Los resultados mostraron una lista ordenada de aldeas, conexiones óptimas y una representación clara en **Turtle**, mejorada para evitar que las líneas interfieran con el texto. Se logró un diseño funcional y eficiente, permitiendo modelar la mejor estrategia de difusión de mensajes entre aldeas con **mínima distancia y máxima cobertura**, replicando el funcionamiento óptimo de una red de palomas mensajeras.