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 basada en un heap binario**, 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).

Se utiliza además un **contador incremental** para preservar el orden de llegada cuando dos pacientes tienen el mismo nivel de riesgo.

Complejidad

- **Inserción (`push`)**: O(log n)
- **Eliminación (`pop`)**: O(log n)

Esta estructura es eficiente incluso con una gran cantidad de pacientes.

```
import heapq

import itertools
```

heapq: módulo de Python que implementa colas de prioridad usando montículos (heaps).

itertools.count(): genera un contador infinito para diferenciar el orden de llegada.

```
class PrioridadQueue:
```

Una **cola de prioridad genérica**, es decir, puede almacenar cualquier tipo de dato (`item`).

```
    def __init__(self):
        self._heap = []
        self._counter = itertools.count()
```

- `_heap`: la estructura base (min-heap).
- `_counter`: se usa para desempatar cuando hay prioridades iguales.

```
  def push(self, item, prioridad):
        count=next(self._counter)
        heapq.heappush(self._heap, (prioridad, count, item))
```

Se inserta una tupla `(prioridad, orden_llegada, item)` en el heap.

El heap usará:

- `prioridad` para determinar el orden de atención.
- `count` para resolver empates (el menor count representa al paciente que llegó antes).

```
  def pop(self):
        if not self._heap:
            raise None
        return heapq.heappop(self._heap)[-1]
```

Extrae y devuelve el `item` de **mayor prioridad** (es decir, menor número de riesgo).

`raise None` no es una práctica válida: debería ser `raise IndexError("La cola está vacía")` o similar.

```
def size(self):
        return len(self._heap)

    def is_empty(self):
        return len(self._heap) == 0
```

`size()` devuelve el número de elementos en la cola.

`is_empty()` indica si ya no quedan pacientes por atender.

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

El objetivo es encontrar la **forma más eficiente de propagar el mensaje** de manera que:

- Cada aldea reciba la noticia **solo una vez**.
- Las rutas usadas minimicen la **suma total de distancias** recorridas por todas las palomas.
- Se determine qué aldea debe recibir la noticia desde cuál, y a qué otras debe reenviarla.

```
import heapq
from collections import defaultdict
```

- `heapq`: utilizado para manejar la cola de prioridad del algoritmo de Dijkstra.
- `defaultdict`: simplifica la creación de diccionarios anidados para representar el grafo.

```
def leer_grafo(filename):
    grafo = defaultdict(list)
    with open(filename, 'r') as archivo:
        for linea in archivo:
            partes = linea.strip().split(',')
            if len(partes) != 3:
                continue  # Ignora líneas vacías o mal formateadas
            origen, destino, distancia = partes
            distancia = int(distancia)
            grafo[origen].append((destino, distancia))
            grafo[destino].append((origen, distancia))  # Grafo no
dirigido
    return grafo
```

Lee el archivo `aldeas.txt`.

Cada línea representa una conexión bidireccional entre dos aldeas y la distancia que las separa.

Construye un **grafo no dirigido** donde las conexiones están representadas como listas de tuplas (vecino, distancia).

```
def dijkstra(grafo, inicio):
    distancias = {nodo: float('inf') for nodo in grafo}
    distancias[inicio] = 0
    predecesores = {nodo: None for nodo in grafo}
    cola = [(0, inicio)]

    while cola:
        distancia_actual, actual = heapq.heappop(cola)

        if distancia_actual > distancias[actual]:
            continue

        for vecino, peso in grafo[actual]:
            nueva_dist = distancia_actual + peso
            if nueva_dist < distancias[vecino]:
                distancias[vecino] = nueva_dist
                predecesores[vecino] = actual
                heapq.heappush(cola, (nueva_dist, vecino))

    return distancias, predecesores
```

- Calcula las **distancias mínimas** desde "Peligros" a todas las otras aldeas.
- Guarda además el **camino óptimo** usando el diccionario `predecesores`, que indica desde qué aldea debe recibirse la noticia.

```
def reconstruir_camino(predecesores, destino):
    camino = []
    while destino is not None:
        camino.append(destino)
        destino = predecesores[destino]
    return camino[::-1]
```

```
A partir del diccionario de predecesores, reconstruye el camino
completo desde "Peligros" hasta una aldea dada.



def imprimir_resultados(grafo, distancias, predecesores):
    for aldea in sorted(grafo):
        if aldea == "Peligros":
            continue

        camino = reconstruir_camino(predecesores, aldea)
        if len(camino) < 2:
            print(f"Aldea: {aldea}\n  Recibe de: -\n  Envía a: -\n")
            continue

        recibe_de = camino[-2]
        envia_a = camino[1]

        print(f"Aldea: {aldea}")
        print(f"  Recibe de: {recibe_de}")
        print(f"  Envía a: {envia_a}\n")

grafo = leer_grafo('data/aldeas.txt')
distancias, predecesores = dijkstra(grafo, "Peligros")
imprimir_resultados(grafo, distancias, predecesores)
```

- Se arma el grafo con los datos de entrada.
- Se aplica Dijkstra desde “Peligros”.
- Se imprimen los resultados.

```
def calcular_distancia_total(distancias):
    return sum(dist for aldea, dist in distancias.items() if aldea !=
"Peligros")

total = calcular_distancia_total(distancias)
print(f"Distancia total recorrida por todas las palomas: {total}
leguas")
```

Calcula el total de leguas recorridas por todas las palomas