Facultad de Ingeniería
UNER

Trabajo N°2 Algoritmos y Estructura de Datos

Alumnos:
Anderson, Mateo.
Costa, Luciana.
Insaurralde, Rosario.

Profesor de comisión: Juan Rizzato

Fecha límite 6 de junio de 2025

Problema 1.

Para simular una sala de emergencias, decidimos usar una cola de prioridad, ya que permite atender primero a los pacientes más críticos sin importar el orden en el que llegaron. Esta estructura representa mejor cómo funciona una sala real, donde no se sigue un orden fijo, sino que se prioriza según la urgencia médica.
Primeramente, se construyó a partir de una lista de Python, en la cual se insertan los pacientes manteniendo un orden basado en su nivel de riesgo (donde 1 = más crítico, 2 = medio y 3 = menos crítico).
Además, si dos pacientes tienen el mismo riesgo, se les da prioridad al que llegó antes, utilizando un número de llegada. Esta comparación doble se programó en el método __lt__() de la clase Paciente.

Complejidad de las operaciones.
Insertar un paciente (agregar): tiene una complejidad de O(log n). Esto se debe a que después de agregar al paciente, se lo posiciona hacia arriba para mantener el orden de prioridad.
Eliminar un paciente (eliminar): también es de O(log n), porque al quitar al paciente más prioritario , se debe reordenar la estructura desplazando elementos hacia abajo.

En conclusión todo esto nos permitió organizar a los pacientes de forma eficiente y justa, algo clave en una sala de emergencias. A diferencia de una lista o una cola común, que atienden por orden de llegada, esta solución respeta el nivel de criticidad médica, lo cual tiene más sentido en un contexto real. Además, las operaciones siguen siendo rápidas incluso con muchos pacientes, lo que mejora el rendimiento del programa.
Y en cuanto a la cola de prioridad fue una buena elección para esta simulación ya que refleja mejor la lógica del problema, mantiene el código organizado y permite trabajar con muchos datos sin afectar el funcionamiento.

Problema 2.

En este problema desarrollamos una base de datos para guardar mediciones de temperatura asociadas a fechas. Para organizar la información de forma eficiente, se usó un árbol binario equilibrado (AVL), que mantiene los datos ordenados por fecha y permite búsquedas rápidas.
La clase principal del módulo es Temperaturas_DB, que permite guardar temperaturas, eliminarlas, consultar temperaturas en un día específico, y también obtener listados, mínimos y máximos dentro de un rango de fechas.

Cada medición está compuesta por una temperatura (número entero en ºC) y una fecha en formato "dd/mm/aaaa" (de tipo objeto Fecha). Inicialmente intentamos trabajar directamente con objetos datetime, pero encontramos varias dificultades para utilizarla, por lo que decidimos realizar una clase llamada Fecha para gestionar estos valores, la misma nos causó dificultades más adelante por lo cual decidimos incluir la función datetime en ella para poder sumar fechas y así trabajar con intervalos.

Los métodos utilizados fueron::

- guardar_temperatura (fecha, temp): guarda una temperatura para una fecha.
- devolver_temperatura (fecha): devuelve la temperatura de una fecha, si está registrada.
- max_temp_rango (fecha1, fecha2) y min_temp_rango(fecha1, fecha2): buscan las temperaturas máxima y mínima entre dos fechas, respectivamente.
- temp_extremos_rango (fecha1, fecha2): devuelve la máxima y la mínima temperatura en un rango de dos fechas.
- devolver_temperaturas fecha1, fecha2): lista todas las mediciones entre dos fechas.
- borrar_temperatura (fecha): elimina una medición registrada.
- cantidad_muestras (): indica cuántas temperaturas hay guardadas en total.

Para las pruebas agregamos varias fechas con diferentes temperaturas, y pudimos lograr eliminar un registro y se verificó que los resultados fueran correctos. También se comprobaron los métodos para buscar valores extremos y recorrer datos por rangos.
Esta solución permite trabajar con un gran volumen de datos sin perder eficiencia, gracias al uso del árbol AVL.

Complejidad de las operaciones.

Método	Complejidad	Análisis
guardar_temperatura(fecha, temperatura)	O(log n)	Convierte la fecha a objeto Fecha (O(1)) y la inserta en el árbol AVL, que tiene complejidad logarítmica al estar balanceado.
devolver_temperatura(fecha)	O(log n)	Convierte la fecha (O(1)) y busca en el árbol AVL, lo cual tiene complejidad logarítmica.
borrar_temperatura(fecha)	O(log n)	Convierte la fecha (O(1)), verifica su existencia (O(log n)) y la elimina del árbol AVL (O(log n)).
devolver_temperaturas(fecha1, fecha2)	O(k log n)	Para cada día en el rango [fecha1, fecha2] se hace una búsqueda en el AVL (O(log n)), con k días en total.
max_temp_rango(fecha1, fecha2)	O(k log n)	Se recorren k días del rango, haciendo una búsqueda O(log n) por cada uno.
min_temp_rango(fecha1, fecha2)	O(k log n)	Igual que max_temp_rango, se recorre el rango y se hace una búsqueda por cada fecha.
temp_extremos_rango(fecha1, fecha2)	O(k log n)	Llama internamente a min_temp_rango y max_temp_rango, ambas O(k log n), por lo que la complejidad total es también O(k log n).
cantidad_muestras()	O(1)	Devuelve el tamaño actual del árbol AVL, valor que se mantiene actualizado constantemente.
borrar()	O(1)	Reemplaza el árbol AVL por uno nuevo vacío; operación constante.
len()	O(1)	Método mágico que devuelve el tamaño actual del árbol AVL.
contains(fecha)	O(log n)	Método mágico que busca una fecha en el árbol AVL, con búsqueda logarítmica.
iter()	O(n)	Método mágico que recorre el árbol completo en orden; n es la cantidad de temperaturas almacenadas.

Problema 3.

El objetivo principal del programa era representar una red de aldeas mediante un grafo y calcular las rutas más cortas desde una aldea de inicio hacia todas las demás, utilizando el algoritmo de Dijkstra. Además, buscamos organizar las aldeas en orden alfabético, mostrando qué aldea recibe y a cuáles envía cada una, y calcular la suma total de distancias recorridas.

El código está dividido en varias partes que se complementan entre sí:

Clase Vértice: esta clase representa una aldea y cada vértice guarda:

- Su nombre (id)
- Las aldeas vecinas conectadas a él y la distancia hacia ellas
- La distancia mínima desde el punto de partida
- El vértice anterior (predecesor) en la ruta óptima

Clase Grafo: se encarga de almacenar todas las aldeas (vértices) y las conexiones entre ellas (aristas). Permite:

- Agregar vértices (aldeas)
- Consultar si un vértice existe
- Obtener un vértice por su nombre

Clase Cola de prioridad: Se emplea en la implementación del algoritmo de Prim, ya que permite seleccionar siempre el vértice no visitado con la menor distancia al árbol de expansión mínima construido hasta el momento.Permite:

- esta_vacia(): devuelve True si la cola no contiene ningún elemento, indicando que ya no hay vértices por procesar.
- agregar(dato):Inserta un nuevo elemento en la cola de prioridad.
- eliminar():Extrae y devuelve el elemento con la menor prioridad de la cola.

Clase MonticuloMin: Esta clase implementa una cola de prioridad basada en un montículo mínimo (min-heap). Se utiliza en el algoritmo de Prim para seleccionar siempre el vértice no visitado con la menor distancia al árbol de expansión mínima construido hasta el momento.Permite:

- insertar(prioridad, vértice):Agrega un nuevo vértice a la cola, junto con su prioridad (por ejemplo, la distancia mínima encontrada hasta ese momento).
- extraer_min():Busca y elimina de la cola el vértice con la menor prioridad, es decir, el que debe ser procesado a continuación.
- esta_vacio():Devuelve True si la cola no contiene ningún elemento, indicando que ya no hay vértices por procesar.

Funciones principales (aplicación):

- construir_grafo_aldeas (path_archivo): Lee un archivo de texto con la lista de aldeas y sus conexiones. Crea los vértices y aristas correspondientes en el grafo.
- aldeas_alfabetico (grafo): Devuelve una lista de nombres de aldeas ordenados alfabéticamente.
- prim (grafo, inicio): Ejecuta el algoritmo de Prim para hallar el árbol de expansión mínima (MST) desde la aldea inicial, conectando todas las aldeas con el menor costo total y sin ciclos. Utiliza una cola de prioridad para seleccionar siempre la arista de menor peso que conecta un nuevo vértice al árbol.
- obtener_rutas (grafo, inicio): Devuelve una lista con la información de qué aldea recibe la noticia de cuál y a quién le envía información, según el árbol de expansión mínima calculado.
- sumar_distancias (grafo, inicio): Suma todas las distancias del árbol de expansión mínima desde la aldea de inicio hacia el resto, es decir, la suma de los pesos de las aristas del MST.

Dentro de la aplicación se ejecuta el main donde se construye el grafo a partir del archivo (aldeas.txt), que contiene aldeas solas (se agregan como vértices) y conexiones entre aldeas (origen, destino, distancia). Luego se muestran las aldeas ordenadas, las rutas óptimas y la suma total de distancias del árbol de expansión mínima.