Informe

A continuación, se presentan las modificaciones realizadas en el código del ejercicio 1 del Trabajo Práctico 2, que involucra la simulación de la atención de pacientes en una cola de espera. Se ha migrado de una implementación basada en un Árbol Binario de Búsqueda (ABB) a una implementación utilizando una Cola de Prioridad basada en un heap.

Cambios Realizados:

Estructura de Datos:

Se reemplazó el Árbol Binario de Búsqueda (ABB) por una Cola de Prioridad implementada como un heap.

La clase PriorityQueue se introdujo para gestionar la cola de espera.

Métodos y Atributos:

Métodos de agregar y eliminar pacientes cambiados para adaptarse a la nueva estructura de datos.

Métodos de búsqueda y obtención de máxima prioridad eliminados.

Se añadieron métodos get_contador y __lt__ en la clase Paciente para comparación y orden en la cola de prioridad.

Flujo Principal:

Se modificó el bucle principal para simular la sala de emergencias utilizando la cola de prioridad.

Se introdujo un contador incremental para asignar a cada paciente.

Espera entre Simulaciones:

Se agregó un intervalo de 1 segundo entre simulaciones y atención de pacientes para simular el tiempo real.

Conclusiones:

Estos cambios proporcionan una implementación más eficiente y simple para la gestión de una sala de emergencias simulada. La migración de un ABB a una Cola de Prioridad mejora la velocidad de inserción y eliminación de pacientes, simplificando el código y haciendo que la simulación sea más realista.

Orden de Complejidad:

Inserción en la Cola de Prioridad (heap): O(log n) en promedio, donde 'n' es la cantidad de pacientes en espera.

Eliminación en la Cola de Prioridad (heap): O(log n) en promedio, donde 'n' es la cantidad de pacientes en espera.

Ejercicio 2: Implementación de un Árbol AVL para el Almacenamiento y Manipulación de Datos de Temperatura

Introducción

El presente informe describe la implementación de una estructura de datos de árbol AVL para el almacenamiento y manipulación de datos de temperatura junto con sus fechas correspondientes. Un árbol AVL es una estructura de datos de árbol binario balanceado que garantiza que la diferencia de alturas entre los subárboles izquierdo y derecho de cualquier nodo sea como máximo 1. Esto permite realizar operaciones de búsqueda y actualización de manera eficiente.

Descripción del Código:

El código implementa dos clases principales:

- NodoAVL: Esta clase representa un nodo en el árbol AVL. Cada nodo contiene la fecha, la temperatura, punteros a los nodos hijos izquierdo y derecho, y la altura del nodo en el árbol.
- TemperaturasDB: La clase TemperaturasDB utiliza un árbol AVL para almacenar y gestionar las temperaturas.

Pruebas y Resultados

El código incluye pruebas de los métodos implementados para demostrar su funcionamiento. A continuación, se presentan los resultados de estas pruebas:

- Inserción de Datos: Se insertan cuatro temperaturas junto con sus fechas en la base de datos.
- Búsqueda de Temperatura por Fecha: Se consulta una temperatura para una fecha específica, y se obtiene la temperatura correctamente.
- Consulta de Temperaturas en un Rango de Fechas: Se consultan y se muestran las temperaturas dentro de un rango de fechas específico.
- Búsqueda de Temperatura Máxima y Mínima en un Rango de Fechas: Se encuentra tanto la temperatura máxima como la mínima en un rango de fechas y se muestran los resultados.
- Búsqueda de Temperaturas Extremas en un Rango de Fechas: Se encuentran tanto la temperatura máxima como la mínima en un rango de fechas y se muestran los resultados.
- Eliminación de Datos: Se elimina una temperatura asociada a una fecha específica y se confirma que se ha eliminado.

- Cantidad de Muestras Almacenadas: Se consulta la cantidad de muestras almacenadas en la base de datos, y se muestra el número.

Cambios realizados:

- Se codificó una clase llamada ArbolAVL, capaz de alojar cualquier tipo de dato en sus nodos.
- Se modificó la clase TemperaturasDB para que contenga dentro un árbol AVL, y los métodos que solicita el enunciado.

Tabla con el análisis del orden de complejidad Big-O para cada uno de los métodos implementados:

Método	**Orden de Complejidad**	**Explicación**
altura	O(1)	La altura de un nodo ya está almacenada en el nodo mismo, por lo que obtenerla es una operación constante
actualizar_altura	O(1)	La actualización de la altura de un nodo también es una operación constante.
balance	O(1)	Calcular el balance de un nodo implica restar las alturas de sus subárboles, una operación constante.
rotar_izquierda	O(1)	Las rotaciones a la izquierdas son operaciones constantes, no dependen del tamaño del árbol.
rotar_derecha	O(1)	Las rotaciones a la derecha son operaciones constantes, no dependen del tamaño del árbol.
insertar	O(log n)	La inserción en un árbol AVL tiene una complejidad de O(log n), donde n es el número de nodos en el árbol,

		debido a las rotaciones que pueden ser necesarias para mantener el equilibrio.
guardar_temperatura	O(log n)	tiene una complejidad de O(log n), ya que llama a insertar, Estas operaciones implican navegar a través del árbol desde la raíz hasta el nodo deseado.
devolver_temperatura	O(log n)	La búsqueda en un árbol AVL tiene una complejidad de O(log N) debido a que se realiza de manera similar a la inserción, descendiendo por el árbol de forma balanceada.
max_temp_rango	O(n * log n)	Donde n es el número de temperaturas en el rango. La búsqueda y comparación para encontrar la temperatura máxima ocurren para cada temperatura en el rango.
min_temp_rango	O(n * log n)	Donde n es el número de temperaturas en el rango. La búsqueda y comparación para encontrar la temperatura mínima ocurren para cada temperatura en el rango.
temp_extremos_rango	O(n * log n)	Donde n es el número de temperaturas en el rango. La búsqueda y comparación para encontrar las temperaturas y ordenarlas de la mínima hasta la

		máxima, ocurren para cada temperatura en el rango.
borrar_temperatura	O(log n)	La eliminación en un árbol AVL tiene una complejidad de O(log N), similar a la inserción.
cantidad_muestras	O(1)	Obtener la cantidad de muestras es una operación constante, ya que el contador de muestras se mantiene actualizado en cada operación de inserción y eliminación

Conclusión:

Hemos desarrollado un sistema de gestión de temperaturas basado en un árbol AVL, lo que garantiza un rendimiento eficiente durante la inserción, búsqueda y eliminación de datos.

Utilizamos estructuras de datos como nodos y árboles para organizar y mantener las temperaturas asociadas a fechas específicas. El algoritmo AVL asegura un equilibrio adecuado del árbol.

Implementamos funciones para guardar, buscar y eliminar temperaturas en la base de datos. Además, manejamos errores de formato de fecha para mantener la integridad de los datos.

Nuestro programa permite consultar temperaturas en rangos específicos de fechas, incluyendo la opción de encontrar temperaturas máximas y mínimas dentro de un período determinado.

Ejercicio 3: Cálculo del Precio Mínimo de Transporte y Cuello de Botella en una Red de Ciudades

Introducción

Este informe presenta una implementación en Python que permite calcular el precio mínimo para transportar bienes desde una ciudad de origen a cualquier destino, además de identificar el cuello de botella en términos de peso máximo admitido en una red de ciudades interconectadas. El código está basado en un grafo que representa las conexiones entre ciudades, con información sobre el peso máximo admitido y el precio de transporte entre ellas.

Descripción del Código

El código se divide en las siguientes secciones:

1. **Definición de Clases**

- Movimiento: Esta clase representa una conexión entre dos ciudades e incluye información sobre el peso máximo admitido y el precio del transporte.
- Vertice: Representa nodos que corresponden a las ciudades. Cada vértice mantiene un seguimiento de si ha sido visitado y almacena las conexiones (movimientos) hacia otras ciudades, junto con su peso máximo admitido y precio.
- Grafo: Esta clase define un grafo que contiene ciudades y sus conexiones. Contiene métodos para agregar vértices al grafo y agregar movimientos para representar rutas entre ciudades.

2. **Lectura de Datos**

- El código lee los datos desde un archivo de texto llamado "rutas.txt". Cada línea del archivo contiene información sobre una conexión entre dos ciudades, incluyendo la ciudad de origen, ciudad de destino, peso máximo admitido y precio del transporte. Si las ciudades no existen como vértices en el grafo, se crean.

  Para que el código funcione, es necesario modificar la línea que contiene la ruta del archivo de datos para que coincida con la ubicación en la computadora del usuario:

ruta del archivo⇒ r'C:\Users\Fernando\Desktop\AyED_2_parte\TP2\Ejercicio3\rutas.txt'

El usuario debe reemplazar esta ruta con la ubicación de su propio archivo "rutas.txt".

3. **Funciones para Cálculos**

   El código incluye tres funciones principales:

- encontrar_cuello_botella: Encuentra el cuello de botella en términos de peso máximo admitido desde la ciudad de origen.

- buscar_precio_minimo: Calcula el precio mínimo para transportar bienes desde la ciudad de origen a cualquier destino, teniendo en cuenta el cuello de botella. Utiliza recursión para explorar todas las rutas posibles.

- precio_minimo: Combina las dos funciones anteriores para encontrar el precio mínimo de transporte desde la ciudad de origen a cualquier destino.

Pruebas y Resultados

El código realiza una prueba con una ciudad de origen llamada "Ciudad Bs.As." y muestra los resultados:

- El precio mínimo para transportar desde "Ciudad Bs.As." a cualquier destino.
- El peso máximo que se puede transportar desde "Ciudad Bs.As." a cualquier otra ciudad de destino.

Cambios realizados:

- Se procuró manejar rutas relativas y detectar rutas de peso de coste máximo y coste mínimo. Se agregó Dijsktra (con algunas modificaciones), para recibir como entrada el grafo, el nodo de origen y el nodo de destino.
- encontrar_precio_minimo: se realizó un ajuste de lógica para calcular correctamente el precio minimo y el peso maximo

Conclusiones

Hemos empleado diversas estructuras de datos, para modelar el grafo y las conexiones entre ciudades.

Implementamos el algoritmo de Dijkstra para calcular las rutas más económicas desde CiudadBs.As. a otras ciudades del grafo, considerando tanto el precio como el peso máximo transportable.

El programa puede cargar datos desde un archivo de texto, asegurando un manejo adecuado de posibles errores, como archivos no encontrados o problemas de lectura.

Optimizamos el algoritmo utilizando una cola de prioridad basada en montículos, mejorando significativamente la eficiencia del proceso de búsqueda del vértice con la distancia mínima en cada iteración.

Además, implementamos mecanismos de manejo de excepciones para capturar posibles errores durante la ejecución del programa, proporcionando mensajes de error descriptivos.