Trabajo Práctico N°2 - ***Documentación de las Actividades***

Integrantes:

- Agustin Facundo Gaitan
- Candelaria Macchi Rojas

Asignatura: Algoritmos y estructuras de datos.

Fecha de Entrega: Lunes 23 de octubre.

Introducción
El presente documento busca informar acerca de los resultados obtenidos durante la realización del primer trabajo práctico de la asignatura algoritmos y estructuras de datos, para ello, primero repasamos cuáles fueron los objetivos a realizar.

1. Sala de Emergencias

El **triaje** es un proceso que permite una gestión del riesgo clínico para poder manejar adecuadamente y con seguridad los flujos de pacientes cuando la demanda y las necesidades clínicas superan a los recursos. El siguiente proyecto de software posee una sencilla simulación de esta situación: enlace a código.
En este proyecto para simplificar existen pacientes con tres niveles de riesgo para su salud: 1: crítico, 2: moderado, 3: bajo. Actualmente la simulación muestra que los pacientes se atienden según el orden en que llegan al centro de salud.

Se solicita:

a. Seleccionar, programar y aplicar una estructura de datos adecuada para almacenar los pacientes conforme ingresan al centro de salud **de modo tal que cuando se atiende un paciente siempre sea aquel cuyo nivel de riesgo es el más delicado** en comparación con el resto de los pacientes que restan por ser atendidos. Si dos pacientes poseen el mismo nivel de riesgo, adoptar un segundo criterio para seleccionar uno de ellos. Recordar que la estructura de datos debe ser genérica, es decir, debe poder almacenar cualquier tipo de dato, y no ser específica para alojar pacientes (separar implementación de aplicación).
b. Fundamentar en el informe, en no más de una página, la estructura seleccionada indicando el orden de complejidad *O* de inserciones y de eliminaciones en la estructura seleccionada.

Observación: Pueden realizarse todas las modificaciones al código que crean necesarias siempre que se respete el propósito del problema planteado.

2. Temperaturas_DB

Kevin Kelvin es un científico que estudia el clima y, como parte de su investigación, debe consultar frecuentemente en una base de datos la temperatura del planeta tierra dentro de un rango de fechas. A su vez, el conjunto de medidas crece conforme el científico registra y agrega **mediciones** a la base de datos.
Una medición está conformada por el valor de temperatura en ºC (flotante) y la fecha de registro, la cual deberá ser ingresada como “dd/mm/aaaa” (string). (Internamente sugerimos emplear objetos de tipo *datetime*).
Ayude a Joe a realizar sus consultas eficientemente implementando una base de datos en memoria principal “**Temperaturas_DB**” que utilice internamente un árbol AVL. La base de datos debe permitir realizar las siguientes operaciones (interfaz de Temperaturas_DB):

guardar_temperatura(temperatura, fecha): guarda la medida de temperatura asociada a la fecha.

devolver_temperatura(fecha): devuelve la medida de temperatura en la fecha determinada.

max_temp_rango(fecha1, fecha2): devuelve la temperatura máxima entre los rangos fecha1 y fecha2 inclusive (fecha1 < fecha2). Esto no implica que los intervalos del rango deban ser fechas incluidas previamente en el árbol.

min_temp_rango(fecha1, fecha2): devuelve la temperatura mínima entre los rangos fecha1 y fecha2 inclusive (fecha1 < fecha2). Esto no implica que los intervalos del rango deban ser fechas incluidas previamente en el árbol.

temp_extremos_rango(fecha1, fecha2): devuelve la temperatura mínima y máxima entre los rangos fecha1 y fecha2 inclusive (fecha1 < fecha2).

borrar_temperatura(fecha): recibe una fecha y elimina del árbol la medición correspondiente a esa fecha.

devolver_temperaturas(fecha1, fecha2): devuelve un listado de las mediciones de temperatura en el rango recibido por parámetro con el formato “dd/mm/aaaa: temperatura ºC”, ordenado por fechas.

cantidad_muestras(): devuelve la cantidad de muestras de la BD.

Adicionalmente realizar las siguientes actividades:

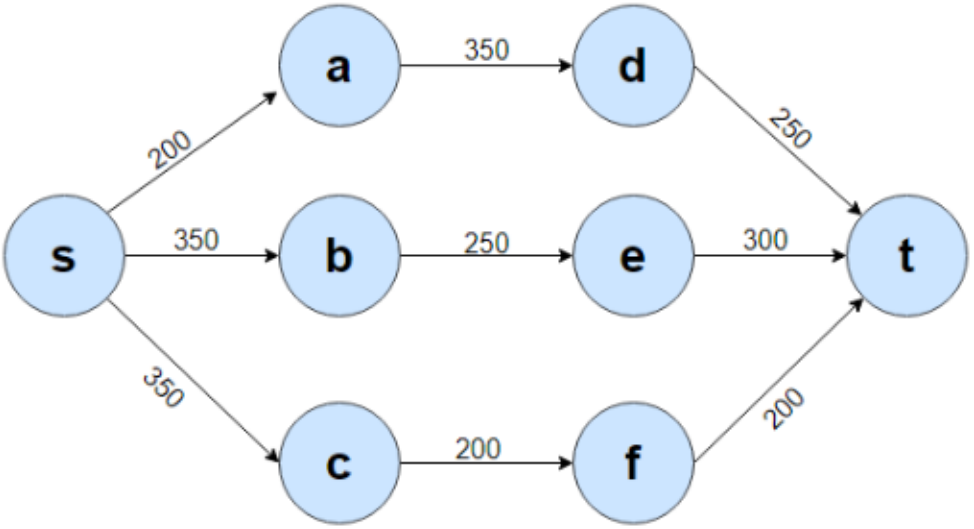
- Escriba una tabla con el análisis del orden de complejidad Big-O para cada uno de los métodos implementados para su clase “Temperaturas_DB”, explique brevemente el análisis de los mismos.

3. Servicio de Transporte

La empresa CasaBella S.A. en Ciudad de Buenos Aires fabrica y vende mobiliario de hogares en distintas ciudades y ha decidido tercerizar la distribución de sus productos. Para esto, contrata un servicio de transporte para llevar sus productos de una ciudad a otra. El responsable de contabilidad de la empresa tiene un archivo con una lista de todas las rutas disponibles, cada ruta en la lista es una tupla (s_i, t_i, w_i, c_i) donde s_i y t_i son respectivamente los nombres (string) de la ciudad de inicio y final de cada ruta de transporte, w_i es un entero positivo que representa la capacidad máxima de peso admitida (en kg) en esa ruta y c_i un entero positivo que representa el precio (en unidades de 1000) de realizar el transporte por la misma (el precio es el mismo si el camión se traslada vacío o a su máxima capacidad). La existencia de una ruta de s_i a t_i no implica la existencia de la ruta t_i a s_i a menos que esté indicado explícitamente.

La capacidad máxima de peso admitida (w_i) en una ruta va a estar limitado por el tramo que tenga menor capacidad de peso en esa ruta, a este valor se le denomina “cuello de botella”.

Este tipo de problema es conocido como “El problema del camino más amplio o del máximo cuello de botella”. Por ejemplo, en el siguiente grafo tenemos 3 rutas (o caminos) entre los vértices “s” y “t” y los pesos en las aristas representan capacidades de peso.



El cuello de botella del primer camino (superior) es 200, porque es el peso de la arista con menor capacidad. Los cuellos de botella de los caminos restantes son 250 y 200 (medio e inferior respectivamente). El máximo cuello de botella del grafo es 250 y será por tanto la máxima capacidad que se podrá transportar entre los vértices “s” y “t”. Alternativamente, si se encontraran dos rutas con el mismo cuello máximo, entonces, se tendrían dos posibles caminos para transportar la misma cantidad de productos.

Ayude a la empresa CasaBella a evaluar sus opciones de transporte implementando un algoritmo que devuelva:

1. El peso máximo w_{max} que se pueda transportar desde la ciudad de Buenos Aires a cualquier otra ciudad de destino.En este ítem, hallar el peso máximo w_{max} que se pueda transportar desde la ciudad de Buenos Aires a una ciudad de destino equivale a encontrar el máximo cuello de botella entre ambas ciudades.
2. El precio mínimo para transportar un mobiliario del peso w_{max} desde la ciudad de Buenos Aires a otra ciudad de destino.

Utilice el archivo provisto en el siguiente enlace para representar el problema planteado y encontrar la solución a los ítems anteriores

Desarrollo

Ejercicio 1: Sala de emergencias

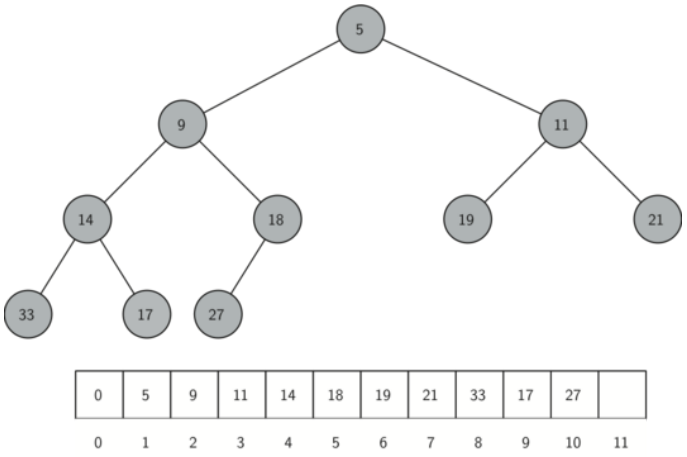
El primer paso para resolver este problema es la elección de la estructura de datos, para ello, determinamos que la mejor opción es un **montículo binario. Recordando, un montículo binario es una ED que se esquematiza como un árbol, a su vez, su lógica puede ser traspolada a una lista de python,** existen dos tipos de montículos,de máximo y de mínimo, en nuestro caso, **el montículo va a ser uno de mínimo, el cual es un montículo en el que las claves más grandes se encuentran siempre hacia el final de**

la cola de prioridad, de manera que los pacientes que tienen un menor nivel de riesgo se acomoden en las ramas superiores y a medida que nos movemos hacia abajo hay pacientes con un mayor nivel de riesgo, si llega un nuevo paciente, debemos acomodarlo en la estructura teniendo en cuenta no sólo la lógica del árbol sino también el segundo criterio el cual será el orden de llegada, pues cada paciente cuando llega se le asigna una fecha y una hora, en nuestro caso a efectos prácticos será un segundo después, para simular aun mejor la sala de hospital, los pacientes se atienden al llegar con una probabilidad del 50%, en caso de que no los atienda, se agregan a la cola de espera.
Esta ED es de gran utilidad para implementar una cola de prioridad, ya que necesitamos poder ordenar bien a los pacientes para ser atendidos según su nivel de riesgo.
La complejidad de agregar o quitar un nuevo elemento al montículo binario (en el peor caso) es **O(Log n), siendo n el número de elementos del montículo**

- **Agregar (Insertar) un elemento:** Cuando se agrega un elemento al montículo, primero se coloca en la parte inferior del árbol (como una hoja) y luego se realiza una operación de "subir" para restaurar la propiedad del montículo. La operación de subir implica comparar y, posiblemente, intercambiar el nuevo elemento con su padre, lo que requiere O(log n) comparaciones y movimientos en el peor caso.
- **Quitar (Eliminar) el elemento máximo o mínimo:** Cuando se elimina el elemento máximo o mínimo de un montículo, se reemplaza por el último elemento en la lista y se realiza una operación de "bajar" para restaurar la propiedad del montículo. La operación de bajar implica comparar y, posiblemente, intercambiar el elemento raíz con uno de sus hijos, lo que también requiere O(log n) comparaciones y movimientos en el peor caso.

Para garantizar un rendimiento logarítmico como ya mencionamos, debemos mantener a nuestro árbol equilibrado. Un árbol binario equilibrado tiene aproximadamente el mismo número de nodos en los subárboles izquierdo y derecho de la raíz. En nuestra implementación del montículo mantenemos al árbol equilibrado creando un árbol binario completo. Un árbol binario completo es un árbol en el que cada nivel tiene todos sus nodos, excepto en el nivel más inferior del árbol

Ejemplo: Montículo binario completo que tiene la propiedad de orden y su lista resultante



Debido a que el árbol está completo, el hijo izquierdo de un padre (**en la posición p**) es el nodo que se encuentra en la posición **2p** en la lista. Del mismo modo, el hijo derecho del padre está en la posición **2p+1** en la lista. Para encontrar el padre de cualquier nodo en el árbol, podemos simplemente usar la división entera de Python. Dado que un nodo esté en la posición **n** en la lista, el padre está en la posición **n/2**

El método que usaremos para almacenar ítems en un montículo depende de **mantener la propiedad de orden del montículo**. La propiedad de orden del montículo es la siguiente: En un montículo, para cada nodo **x** con padre **p** , la clave en **p** es **mayor o igual** a la clave en **x.**

Nota: Al comienzo, estábamos usando un montículo de máximo, pero tuvimos dificultades con el problema dado, por lo que optamos por utilizar uno de mínimo.

monticulo.py

Puesto que todo el montículo binario puede ser representado por una sola lista, todo lo que el constructor hará es inicializar la lista. Un montículo binario vacío tiene un cero solo como primer elemento de **listaMonticulo** este cero no se usa, pero está ahí para que la división entera simple pueda usarse en métodos posteriores.

```
class Monticulo_Min:
    def __init__(self):
        self.listaMonticulo = [0]
```

Inicialización de la clase Monticulo_Min

Métodos que implementamos en nuestro montículo de mínimo:

insertar(self, k): Este método se utiliza para insertar un nuevo elemento k en el montículo. El elemento se agrega al final de la lista y luego se llama al método subir para ajustar la posición del elemento recién insertado, asegurándose de que el montículo mantenga su propiedad de montículo mínimo.

eliminarMin(self): Este método se utiliza para eliminar y devolver el elemento mínimo del montículo. Primero se verifica si el montículo no está vacío. Si no está vacío, se toma el elemento mínimo (que está en la posición 1) y se lo reemplaza por el último elemento en el montículo. Luego, se llama al método bajar para reajustar el montículo y mantener su propiedad de montículo mínimo. Finalmente, se devuelve el elemento mínimo que se eliminó.

estaVacio(self): Este método verifica si el montículo está vacío. Devuelve True si la longitud de listaMonticulo es igual a 1, lo que significa que sólo contiene el marcador cero.

tamanio(self): Este método devuelve el tamaño del montículo, que es igual a la longitud de listaMonticulo menos 1 (excluyendo el marcador cero).

subir(self, i): Este método se utiliza para ajustar la posición de un elemento en el montículo hacia arriba, si es necesario. Comienza en la posición i y se mueve hacia arriba intercambiando elementos hasta que se restablezca la propiedad de montículo mínimo.

bajar(self, i): Este método se utiliza para ajustar la posición de un elemento en el montículo hacia abajo, si es necesario. Comienza en la posición i y se mueve hacia abajo intercambiando elementos con su hijo más pequeño hasta que se restablezca la propiedad de montículo mínimo.

encontrarHijoMinimo(self, i): Este método se utiliza para encontrar el índice del hijo más pequeño de un nodo en el montículo. Compara los elementos en las posiciones i * 2 y i * 2 + 1 y devuelve el índice del hijo con el valor más pequeño.

paciente.py

El único cambio que hicimos fue implementar el método conocido como **__it__** . Define la forma en que se compara un paciente con otro cuando se utiliza el operador de comparación <, que se utiliza para determinar si un paciente es "menor" que otro en un contexto de comparación. En este caso, el método __lt__ se implementa para definir cómo se debe ordenar una lista de pacientes en función de su nivel de riesgo y su hora de llegada. Primero se comparan los niveles de riesgo de dos pacientes, si uno es mayor que otro devuelve true o false, y en caso de que tengan el mismo nivel de riesgo, se los compara por el orden de llegada.

En el código proporcionado, main.py, hemos hecho las siguientes modificaciones:

- **Cambiamos cola_de_espera para que sea una instancia de Monticulo_Max.**
- **En lugar de agregar pacientes a una lista, utilizamos el método insertar de cola_de_espera para agregar pacientes a la cola de prioridad. Los pacientes se agregarán según su nivel de riesgo, de modo que el más crítico esté en la parte inferior del montículo, y se acomoden según la hora de llegada si tienen el mismo nivel de riesgo.**
- **Al atender a un paciente, usamos el método eliminarMin para seleccionar al paciente más crítico de la cola de prioridad.**

Ejercicio 2: Temperaturas_DB

*** NodoAVL.py: ***

Clase NodoAVL:

__init__(self, clave, valor, padre=None): El constructor de la clase recibe tres parámetros: **clave** (la clave que identifica el nodo), **valor** (el valor asociado a la clave) y **padre** (el nodo padre en el árbol, por defecto es None si el nodo es la raíz). El constructor inicializa varios atributos del nodo:

- *clave*: La clave del nodo.
- *valor*: El valor asociado a la clave.
- *izquierdo y derecho*: Los nodos hijos izquierdo y derecho, inicializados como None.
- *padre*: El nodo padre, que se proporciona como argumento o es None por defecto.
- *altura:* La altura del nodo, que se inicia en 1 (ya que es un nodo recién creado).
- *factorEquilibrio*: El factor de equilibrio del nodo, que se inicia en 0.

- **tieneHijoIzquierdo(self):** Este método verifica si el nodo tiene un hijo izquierdo.
- **tieneHijoDerecho(self)**: Similar al método anterior, verifica si el nodo tiene un hijo derecho.
- **esHijoIzquierdo(self)**: Verifica si el nodo es un hijo izquierdo en relación con su nodo padre.
- **esHijoDerecho(self)**: Similar al método anterior, verifica si el nodo es un hijo derecho en relación con su nodo padre.
- **esRaiz(self)**: Verifica si el nodo es la raíz del árbol, es decir, no tiene un nodo padre.

*** AVL.py: ***

Clase AVL:

Tiene los siguientes atributos:
raíz: El nodo raíz del árbol AVL.
tamanio: La cantidad de nodos en el árbol.

Métodos públicos:

longitud(self): Devuelve el tamaño del árbol, es decir, la cantidad de nodos.

insertar(self, clave, valor): Inserta un nuevo nodo con una clave y valor dados en el árbol.

buscar(self, clave): Busca un nodo en el árbol por su clave y devuelve el nodo si lo encuentra.

inorder(self, nodo=None): Recorre el árbol en orden, generando nodos en el orden correcto.

Métodos privados:

_agregar(self, clave, valor, nodoActual): Método interno para agregar un nuevo nodo al árbol. Utiliza recursión para encontrar la ubicación correcta para el nuevo nodo y ajustar el equilibrio del árbol.

_eliminar(self, clave, nodo): Método interno para eliminar un nodo del árbol por su clave. También ajusta el equilibrio del árbol después de la eliminación.

_min_nodo(self, nodo): Encuentra y devuelve el nodo con la clave más pequeña en el subárbol con raíz en el nodo dado.

actualizarEquilibrio(self, nodo): Actualiza el equilibrio del árbol después de la inserción o eliminación de un nodo, propagando los cambios hacia arriba en el árbol.

rotarIzquierda(self, nodo): Realiza una rotación a la izquierda en el nodo dado para restablecer el equilibrio.

rotarDerecha(self, nodo): Realiza una rotación a la derecha en el nodo dado para restablecer el equilibrio.

reequilibrar(self, nodo): Reequilibra el árbol si es necesario, ajustando las alturas y realizando rotaciones según sea necesario.

factorEquilibrio(self, nodo): Calcula el factor de equilibrio de un nodo, que es la diferencia entre la altura del subárbol izquierdo y derecho.

actualizarAltura(self, nodo): Actualiza la altura de un nodo en función de la altura de sus hijos.

Temperaturas_DB.py

__init__(self): El constructor de la clase inicializa una instancia de la clase AVL llamada avl para gestionar las temperaturas y establece el tamaño inicial en 0.

guardar_temperatura(self, temperatura, fecha): Este método permite agregar una temperatura asociada a una fecha específica al árbol AVL. La fecha se convierte de una cadena de texto al formato de fecha utilizando la función datetime.strptime, y luego se inserta en el árbol AVL.

devolver_temperatura(self, fecha): Este método busca una temperatura asociada con una fecha dada en el árbol AVL. La fecha se convierte al formato de fecha y se utiliza el método buscar de la clase AVL para buscar la fecha en el árbol. Si se encuentra, se devuelve la temperatura almacenada; de lo contrario, se devuelve None.

max_temp_rango(self, fecha1, fecha2): Este método devuelve la temperatura máxima dentro de un rango de fechas. Se convierten las fechas al formato de fecha y se llama al método _max_temp_rango para realizar la búsqueda.

_max_temp_rango(self, fecha1, fecha2, nodo): Este es un método privado que realiza una búsqueda recursiva para encontrar la temperatura máxima dentro del rango de fechas dado. Se busca en el subárbol con raíz en el nodo actual y se devuelve la temperatura máxima.

min_temp_rango(self, fecha1, fecha2): Similar al método max_temp_rango, pero busca la temperatura mínima dentro del rango de fechas.

_min_temp_rango(self, fecha1, fecha2, nodo): Similar al método _max_temp_rango, pero busca la temperatura mínima dentro del rango de fechas.

_temp_extremos_rango(self, fecha1, fecha2, nodo=None): Este método llama a los métodos min_temp_rango y max_temp_rango y devuelve una tupla con las temperaturas mínima y máxima dentro del rango de fechas.

borrar_temperatura(self, fecha): Este método permite eliminar una temperatura asociada a una fecha específica del árbol AVL. La fecha se convierte al formato de fecha y se utiliza el método eliminar de la clase AVL para eliminar el nodo correspondiente.

devolver_temperaturas(self, fecha1, fecha2): Este método devuelve una lista de temperaturas dentro de un rango de fechas. Se convierten las fechas al formato de fecha y se llama al método _devolver_temperaturas para realizar una búsqueda inorden en el árbol y recopilar las temperaturas dentro del rango.

_devolver_temperaturas(self, fecha1, fecha2, nodo): Este es un método privado que realiza una búsqueda inorden en el árbol para recopilar las temperaturas dentro del rango de fechas.

cantidad_muestras(self): Devuelve la cantidad de muestras de temperaturas almacenadas en el árbol AVL, que es igual al tamaño del árbol.

Ejercicio 3: Servicio de Transporte

Vertice.py

agregar_vecino(self, vecino, ponderacion=0, atributo_extra=0): Agrega un vecino al vértice actual. Puedes proporcionar una ponderación y un atributo extra para la conexión. Estos valores se almacenan en el diccionario **conectado_a**, donde la clave es el vecino y el valor es otro diccionario que contiene la ponderación y el atributo extra. Estos dos representan peso y precio respectivamente, pero se eligieron esos nombres para mantener la generalidad del código.

__str__(self): Define la representación en cadena del objeto. Devuelve una cadena que muestra el ID del vértice y una lista de IDs de los vértices a los que está conectado.

obtener_conexiones(self): Devuelve las claves (IDs) de los vértices vecinos.

obtener_id(self): Devuelve el ID del vértice.

__iter__(self): Permite la iteración sobre los vecinos del vértice.

obtener_ponderacion(self, vecino): Devuelve la ponderación de la conexión entre el vértice actual y su vecino.

obtener_atributo_extra(self, vecino): Devuelve el atributo extra de la conexión entre el vértice actual y su vecino.

obtener_precio(self): Devuelve el precio almacenado en el vértice.

obtener_peso(self): Devuelve el peso almacenado en el vértice.

asignar_precio(self, nuevo_precio): Asigna un nuevo valor al precio del vértice.

asignar_peso(self, nuevo_peso): Asigna un nuevo valor al peso del vértice.

asignar_predecesor(self, predecesor): Asigna un predecesor al vértice, útil en algoritmos como Dijkstra o Bellman-Ford.

grafo.py

agregar_vertice(self, vértice): Este método agrega un vértice al grafo. Toma un objeto Vértice como argumento y lo agrega al diccionario vértices utilizando el id del vértice como clave.

obtener_vertice(self, clave): Este método devuelve el vértice correspondiente a la clave proporcionada. Si la clave no está presente en el diccionario de vértices, devuelve None.

agregar_arista(self, desde, hacía, peso, atributo_extra=0): Este método agrega una arista al grafo entre los vértices con claves desde y hacia. Si los vértices no existen, se crean antes de agregar la arista. El parámetro peso representa el peso de la arista, y

atributo_extra es un atributo opcional que se puede asociar con la arista. La arista se agrega al vértice correspondiente utilizando el método agregar_vecino de la clase Vertice (que no se proporciona en el código, pero se asume que existe).

Monticulo_min (Uso en djkstra)

La única diferencia que hay con el ejercicio uno, es el agregado del nuevo método:

decrementarClave(self, vertice, nueva_distancia): Este método decrementa la distancia asociada al vértice especificado y reorganiza el montículo para mantener su propiedad. Itera sobre la lista hasta encontrar el vértice deseado, actualiza la distancia si es menor y luego llama a la función subir para restaurar la propiedad del montículo.

Monticulo_max (Uso en el dijkstra_max_cuello)

insertar(self, k): Inserta un elemento k en el montículo máximo. Agrega el elemento al final de la lista y luego llama a la función subir para restaurar la propiedad del montículo.

eliminarMax(self): Elimina y devuelve el elemento máximo (valor máximo y elemento asociado) del montículo máximo. Si el montículo no está vacío, obtiene el máximo en la posición 1, elimina el último elemento de la lista, coloca el último elemento en la cima y llama a la función bajar para restaurar la propiedad del montículo.

estaVacio(self): Verifica si el montículo máximo está vacío. Devuelve True si el montículo está vacío (tiene un solo elemento ficticio), False en caso contrario.

tamanio(self): Devuelve el número de elementos en el montículo máximo, excluyendo el elemento ficticio en la posición 0.

subir(self, i): Se llama cuando se inserta un elemento en la última posición de la lista. Realiza un intercambio entre el elemento en la posición i y su padre hasta que se restaura la propiedad del montículo.

bajar(self, i): Se llama cuando se elimina el máximo. Mueve el elemento en la posición i hacia abajo en la lista, intercambiándolo con su hijo más grande hasta que se restaura la propiedad del montículo.

encontrarHijoMaximo(self, i): Encuentra el índice del hijo más grande del elemento en la posición i. Compara los valores directamente en lugar de las distancias.

dijkstra.py

dijkstra_max_cuello(grafo, inicio): Implementa el algoritmo de Dijkstra para encontrar el camino de cuello de botella máximo desde el vértice de inicio en el grafo. Utiliza un montículo máximo para seleccionar el vértice con el cuello de botella máximo en cada iteración.

dijkstra(grafo, inicio): Implementa el algoritmo de Dijkstra estándar para encontrar los caminos más cortos desde el vértice de inicio en el grafo. Utiliza un montículo mínimo para seleccionar el vértice con la distancia mínima en cada iteración

imprimir_rutas_precio(grafo, ciudad_inicial): Imprime los resultados de los caminos más cortos en términos de precio (costo). Itera sobre todos los vértices del grafo y muestra el camino mínimo desde la ciudad inicial a cada destino, junto con el precio mínimo.

imprimir_rutas_peso(grafo, ciudad_inicial): Imprime los resultados de los caminos más cortos en términos de peso admitido. Itera sobre todos los vértices del grafo y muestra el camino mínimo desde la ciudad inicial a cada destino, junto con el peso máximo admitido.