Facultad de Ingeniería

Trabajo Práctico N°2

Algoritmos y Estructuras de Datos

Profesores: Dr. Javier Eduardo Diaz Zamboni, Nicolet, Jonathan

Integrantes: Luis Alberto Rodriguez, Luis Diego Rodriguez

SALA DE EMERGENCIAS

- La Clase **MonticuloBinario**: se utiliza para implementar un montículo binario, que es una estructura de datos similar a un árbol binario y la utilizamos para implementar colas de prioridad.

    - En el constructor **__init__**, iniciamos una lista llamada **listaMonticulo** con un valor inicial de [0] y se inicializa **tamanoActual** en 0. La lista se usa para almacenar los elementos del montículo, y **tamanoActual** rastrea el número de elementos.
    - El método **infiltArriba** se utiliza para mover un elemento hacia arriba en el montículo. Se ejecuta un bucle mientras el elemento actual sea menor que su padre, y se intercambian si es necesario.
    - El método **insertar** agrega un elemento al montículo, lo coloca al final de la lista y luego llama a **infiltArriba** para garantizar que el montículo siga siendo válido.
    - El método **infiltAbajo** se utiliza para mover un elemento hacia abajo en el montículo. Compara el elemento actual con sus hijos y lo intercambia con el hijo más pequeño si es necesario.
    - El método **hijoMin** devuelve la posición del hijo más pequeño de un nodo en el montículo.
    - El método **eliminarMin** elimina y devuelve el elemento más pequeño del montículo.
    - El método **construirMonticulo** construye un montículo a partir de una lista no ordenada de elementos.
    - El método **estaVacia** verifica si el montículo está vacío.

- La clase **MonticuloBinarioPaciente**: utiliza un objeto de la clase **MonticuloBinario** para gestionar pacientes en una sala de emergencias.

    - En el constructor **__init__**, se crea una instancia de **MonticuloBinario** llamada **monticulo** para almacenar a los pacientes.
    - El método **ingresar_paciente** permite ingresar un paciente en el montículo llamando al método insertar del montículo.

- El método **atender** permite atender al paciente con el riesgo más alto, eliminándolo del montículo.
- La clase implementa iteración con los métodos **__iter__** y **__next__** para recorrer los pacientes en orden de prioridad.
- El método **__len__** devuelve el número de pacientes en la sala de emergencias

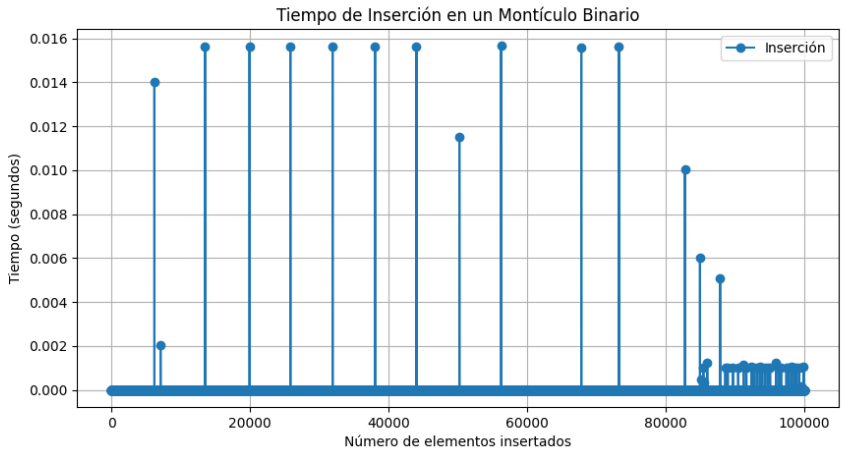
- La clase Paciente: representa a un paciente en la sala de emergencias.

  - En el constructor **__init__**, se generan aleatoriamente un nombre, apellido, nivel de riesgo y descripción de riesgo para el paciente. Los nombres y apellidos se seleccionan aleatoriamente de listas predefinidas, y el nivel de riesgo se elige basado en probabilidades.
  - La clase proporciona métodos para acceder a los atributos del paciente, como nombre, apellido, nivel de riesgo y descripción de riesgo.
  - También se sobrecargan los operadores ==, < y > para comparar pacientes.

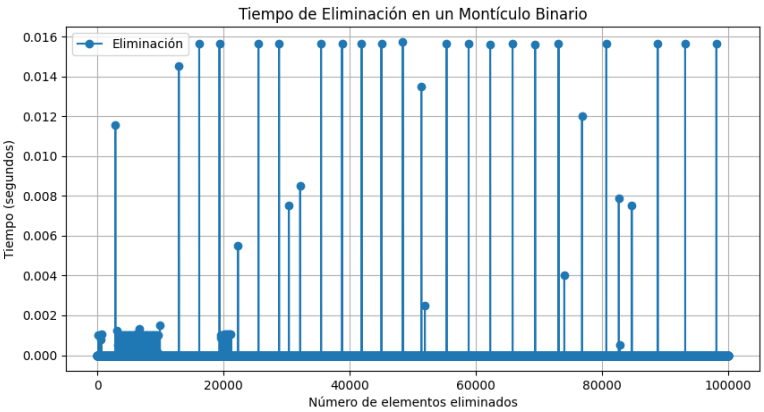
Cuando se realizan Inserciones en el Montículo Binario es complejidad (O(log n)): Cuando se agrega un elemento a un Montículo Binario, inicialmente se coloca en la última posición del montículo (un nodo hoja) y luego se realiza una operación llamada "infiltración hacia arriba". Durante esta operación, el elemento se compara con su padre y se intercambia si es necesario para restaurar la propiedad del montículo (el nodo padre tiene una prioridad mayor/menor que sus hijos). Luego, se repite este proceso mientras se sube en el montículo hasta que se cumple la propiedad del montículo. La operación de infiltración hacia arriba tiene un tiempo de ejecución de O(log n) en el peor caso, ya que un elemento puede necesitar subir hasta la raíz, y el montículo es un árbol binario completo.

Cuando se realizan eliminaciones en el Montículo Binario es complejidad (O(log n)): Cuando se elimina un elemento del Montículo Binario, normalmente se extrae el elemento de máxima (o mínima) prioridad, que se encuentra en la raíz. Luego, el último elemento del montículo se coloca en la raíz y se realiza una operación llamada "infiltración hacia abajo". Durante esta operación, el elemento en la raíz se compara con sus hijos y se intercambia con el hijo que tiene la mayor (o menor) prioridad. Nuevamente, este proceso se repite mientras desciende en el montículo para mantener la propiedad del

montículo. La operación de infiltración hacia abajo tiene un tiempo de ejecución de O(log n) en el peor caso.



Los picos en el tiempo de ejecución pueden estar relacionados con datos particulares que causan cambios en la organización del montículo. En un Montículo Binario, los picos en el tiempo de ejecución podrían ocurrir cuando se inserta un elemento que requiere una reorganización significativa de la estructura para mantener la propiedad del montículo.



Los picos en el tiempo de ejecución pueden estar relacionados con datos particulares que causan cambios en la organización del montículo. En un Montículo Binario, los picos en el tiempo de ejecución podrían ocurrir cuando se elimina un elemento que requiere una reorganización significativa de la estructura para mantener la propiedad del montículo.

Este código representa un sistema de gestión de una sala de emergencias donde se utilizan montículos binarios para administrar a los pacientes según su nivel de riesgo. La clase Paciente se encarga de crear pacientes con datos aleatorios, como nombres y niveles de riesgo. La clase MonticuloBinarioPaciente

actúa como la interfaz principal para ingresar y atender a los pacientes, asegurando que los pacientes más críticos sean atendidos primero. La clase MonticuloBinario proporciona la estructura de datos necesaria para organizar a los pacientes en función de su prioridad en el montículo binario.

TEMPERATURAS_DB

- La Clase ABB: Esta clase representa un Árbol Binario de Búsqueda. Sus características más importantes:

    - En el constructor **__init__**, se inicializa el árbol con una raíz vacía y un tamaño de 0.
    - El método **longitud** y **__len__** permiten obtener la cantidad de nodos en el árbol.
    - El método **agregar** se utiliza para insertar un nodo con una clave y un valor en el árbol. Si el árbol ya tiene una raíz, se llama al método **_agregar** para realizar la inserción de forma recursiva.
    - El método **_agregar** realiza la inserción recursiva en el árbol según la clave. Si la clave es menor que la clave del nodo actual, se va al hijo izquierdo; de lo contrario, se va al hijo derecho.
    - Los métodos **__setitem__** y **__getitem__** permiten establecer y obtener valores en el árbol utilizando la notación de diccionario.
    - Los métodos **obtener** y **_obtener** Estos métodos permiten buscar un nodo en el árbol según su clave. El método **_obtener** se utiliza de manera recursiva para buscar el nodo en el árbol.
    - El método **eliminar** se utiliza para eliminar un nodo según su clave. El método **_eliminar** se llama para llevar a cabo la eliminación.
    - El método **__ delitem __** sobrecarga el operador del para eliminar un nodo utilizando la notación de diccionario.
    - El método **remover** se utiliza para eliminar un nodo del árbol y realizar los ajustes necesarios en la estructura del árbol.
    - El método **__contains__** este método verifica si un nodo con la clave dada existe en el árbol.
    - La clase **NodoArbol** se utiliza para representar los nodos del árbol, con métodos para verificar si un nodo tiene hijos, si es hoja, si es raíz, y otros.

- La clase **NodoArbol:** Es una implementación de un nodo que se utiliza en la construcción de estructuras de datos basadas en árboles, como árboles binarios de búsqueda o árboles AVL. Cada nodo contiene una clave, que generalmente se utiliza para ordenar los nodos en el árbol, y un valor asociado. Los atributos principales de la clase NodoArbol son los siguientes:

    - **clave**: Representa la clave del nodo, que suele ser un valor único utilizado para organizar los nodos en el árbol.
    - **cargaUtil**: Representa el valor asociado a la clave del nodo. Este valor puede ser cualquier tipo de información que se desee almacenar en el nodo.
    - **hijoIzquierdo** y **hijoDerecho**: Estos atributos apuntan a los nodos hijos izquierdo y derecho del nodo actual. Estos hijos son nodos que tienen claves menores o mayores que la clave del nodo actual, respectivamente. Estos atributos permiten la estructura jerárquica de los árboles.
    - **padre**: Este atributo apunta al nodo padre del nodo actual. Permite navegar desde un nodo a su nodo padre en la jerarquía del árbol.
    - **factorEquilibrio**: El atributo **factorEquilibrio** se usa comúnmente en árboles AVL para mantener un seguimiento del equilibrio del árbol. El equilibrio es una propiedad importante en los árboles AVL para garantizar un rendimiento eficiente en las operaciones de búsqueda, inserción y eliminación.

Además de estos atributos, la clase NodoArbol proporciona una serie de métodos que son útiles al trabajar con árboles. Estos métodos incluyen:

- **tieneHijoIzquierdo()** y **tieneHijoDerecho()**: Verifican si el nodo tiene un hijo izquierdo o derecho.
- **esHijoIzquierdo()** y **esHijoDerecho()**: Comprueban si el nodo es un hijo izquierdo o derecho en relación con su nodo padre.
- **esRaíz()**: Indica si el nodo es la raíz del árbol, es decir, si no tiene un nodo padre.
- **esHoja()**: Comprueba si el nodo es una hoja, es decir, si no tiene hijos.
- **tieneAlgunHijo()**: Verifica si el nodo tiene al menos un hijo (izquierdo o derecho).

- **tieneAmbosHijos()**: Comprueba si el nodo tiene tanto un hijo izquierdo como un hijo derecho.
- **reemplazarDatoDeNodo**(clave, valor, hizq, hder): Reemplaza los atributos del nodo (clave, valor, hijos izquierdo y derecho) con los valores especificados.
- **encontrarSucesor()**: Encuentra el sucesor inmediato del nodo en el árbol, que es el nodo con la clave más pequeña que es mayor que la clave del nodo actual.
- **encontrarMin()**: Encuentra el nodo con la clave más pequeña en el subárbol con raíz en el nodo actual.
- **empalmar()**: Este método se utiliza para eliminar un nodo del árbol sin romper la estructura del árbol. Si el nodo es una hoja, simplemente lo elimina. Si el nodo tiene un solo hijo, el hijo se conecta al padre del nodo. Este proceso asegura que la estructura del árbol se mantenga intacta.
- **__iter__()**: Permite la iteración sobre los nodos del subárbol con raíz en el nodo actual en orden ascendente de claves.

- La clase AVL: Esta clase hereda de **ArbolBinarioBusqueda** los métodos y agrega características relacionadas con el equilibrio del árbol:

  - El constructor **__init__** llama al constructor de la clase base(**ArbolBinarioBusqueda**).
  - El método **agregar** ha sido modificado para mantener el equilibrio del árbol AVL. Cuando se agrega un nuevo nodo, se verifica y ajusta el **factor de equilibrio** en los nodos del camino hacia la raíz.
  - El método **actualizarEquilibrio** se encarga de actualizar el factor de equilibrio de los nodos del árbol.
  - Los métodos **rotarIzquierda** y **rotarDerecha** se utilizan para realizar rotaciones en el árbol y mantener el equilibrio.
  - El método **reequilibrar** verifica y ajusta el equilibrio del árbol en caso de desequilibrio.
  - El método mostrar permite imprimir el árbol de forma legible.

- La clase **AVL_Implementacion**: Esta clase actúa como una interfaz para gestionar un árbol **AVL** y agregar características específicas:

- El método **guardar_temperatura** permite agregar una temperatura asociada a una fecha en el árbol AVL.
- El método **devolver_temperatura** permite obtener la temperatura asociada a una fecha dada.
- Los métodos **max_temp_rango** y **min_temp_rango** permiten encontrar las temperaturas máximas y mínimas en un rango de fechas.
- La función **obtener_rango** se encarga de encontrar y devolver las temperaturas almacenadas en el rango de fechas proporcionado. Este rango de fechas se especifica utilizando dos parámetros: fecha_inicio y fecha_fin. Estas fechas se pasan en formato de cadena y se convierten en objetos datetime para su posterior comparación. El método **_obtener_rango** es un método auxiliar que se utiliza de manera recursiva para recorrer el árbol AVL. Comienza desde la raíz del árbol y examina cada nodo para verificar si su clave (representando una fecha) se encuentra dentro del rango especificado. Si es así, se añade al resultado junto con su valor de temperatura.
- El método **temp_extremos_rango** encuentra tanto la temperatura máxima como la mínima en un rango de fechas.
- El método **devolver_temperaturas** permite obtener todas las temperaturas en un rango de fechas.
- El método **cantidad_muetras** devuelve la cantidad de muestras en el árbol.
- El método **borrar_temperatura** permite eliminar una temperatura asociada a una fecha específica del árbol.

Método	Complejidad (Big-O)	Descripción
guardar_temperatura	O(log n)	Agrega una temperatura al árbol AVL, que en el peor caso recorre la altura del árbol.
devolver_temperatura	O(log n)	Obtiene una temperatura por fecha del árbol AVL, que en el peor caso recorre la altura del árbol.

max_temp_rango	O(log n)	Busca la temperatura máxima en un rango de fechas, que en el peor caso recorre la altura del árbol.
obtener_rango	O(log n)	Obtiene una lista de temperaturas en un rango de fechas, que en el peor caso recorre la altura del árbol.
_obtener_rango	O(n)	Método auxiliar que realiza una búsqueda en un rango de fechas, recorriendo el árbol.
min_temp_rango	O(log n)	Busca la temperatura mínima en un rango de fechas, que en el peor caso recorre la altura del árbol.
temp_extremos_rango	O(log n)	Busca la temperatura máxima y mínima en un rango de fechas, que en el peor caso recorre la altura del árbol.
devolver_temperaturas	O(log n)	Obtiene una lista de temperaturas en un rango de fechas, que en el peor caso recorre la altura del árbol.
cantidad_muestras	O(1)	Devuelve la cantidad de muestras en el árbol, lo que es una operación de tiempo constante.
borrar_temperatura	O(log n)	Elimina una temperatura del árbol AVL, que en el peor caso recorre la altura del árbol y realiza

En resumen, este código crea una estructura de datos de árbol que permite el almacenamiento y gestión de datos relacionados con temperaturas y fechas, manteniendo la propiedad de búsqueda en orden y equilibrio en el caso de un árbol AVL. Las clases y métodos proporcionados permiten realizar diversas operaciones para agregar, buscar y eliminar datos de manera eficiente.

SERVICIO DE TRANSPORTE

- La clase **ColaPrioridadMax** representa una cola de prioridad máxima. En una cola de prioridad máxima, los elementos con el valor más alto tienen la prioridad más alta y se eliminan antes que los elementos con un valor menor. Esta clase es utilizada en el algoritmo Dijkstra con el cuello de botella máximo para mantener un seguimiento de los vértices con la máxima prioridad. Los métodos son los siguientes:

    - **__init__(self)**: Constructor que inicializa una lista para almacenar las tuplas (clave, valor) y el tamaño del montículo. Inicializa la lista de montículo con una tupla ficticia y el tamaño a 0.
    - **construirMonticulo(self, lista)**: Método para construir un montículo a partir de una lista de elementos (claves, valores). Establece el tamaño del montículo y la lista de montículo con la lista de elementos, inicializando previamente con una tupla ficticia.
    - **eliminarMax(self)**: Elimina el elemento con la máxima prioridad (valor) y lo devuelve. Actualiza la estructura de datos del montículo.
    - **decrementarClave(self, nodo, nueva_prioridad)**: Decrementa la prioridad (clave) de un nodo específico en el montículo. Actualiza la estructura de datos del montículo y reorganiza si es necesario.
    - **estaVacia(self)**: Verifica si el montículo está vacío.
    - **subir(self, i)**: Ajusta el montículo subiendo un nodo específico para mantener la propiedad del montículo máximo.
    - **bajar(self, i)**: Ajusta el montículo bajando un nodo específico para mantener la propiedad del montículo máximo.
    - **encontrarMaxHijo(self, i)**: Encuentra el hijo con máxima prioridad del nodo en la posición 'i'.

- La clase **ColaPrioridadMin** representa una cola de prioridad mínima. En una cola de prioridad mínima, los elementos con el valor más bajo tienen la prioridad más alta y se eliminan antes que los elementos con un valor mayor. Esta clase se utiliza en el algoritmo Dijkstra para mantener un seguimiento de los vértices con la mínima prioridad. Los métodos son los siguientes:

- **__init__(self)**: Constructor que inicializa una lista para almacenar las tuplas (clave, valor) y el tamaño del montículo. Inicializa la lista de montículo con una tupla ficticia y el tamaño a 0.
- **construirMonticulo(self, lista)**: Método para construir un montículo a partir de una lista de elementos (claves, valores). Establece el tamaño del montículo y la lista de montículo con la lista de elementos, inicializando previamente con una tupla ficticia.
- **eliminarMin(self)**: Elimina el elemento con la mínima prioridad (valor) y lo devuelve. Actualiza la estructura de datos del montículo.
- **decrementarClave(self, nodo, nueva_prioridad)**: Decrementa la prioridad (clave) de un nodo específico en el montículo. Actualiza la estructura de datos del montículo y reorganiza si es necesario.
- **estaVacia(self)**: Verifica si el montículo está vacío.
- **subir(self, i)**: Ajusta el montículo subiendo un nodo específico para mantener la propiedad del montículo mínimo.
- **bajar(self, i)**: Ajusta el montículo bajando un nodo específico para mantener la propiedad del montículo mínimo.
- **encontrarMinHijo(self, i)**: Encuentra el hijo con mínima prioridad del nodo en la posición 'i'.

- La clase **Grafo** representa un grafo dirigido utilizando la representación de lista de adyacencia. Contiene métodos para agregar vértices, agregar aristas, buscar vértices, construir un grafo basado en rutas filtradas, encontrar caminos posibles, eliminar rutas con sobrepeso, encontrar cuellos de botella en rutas posibles y otros métodos relacionados con la manipulación de un grafo. Los métodos son los siguientes:

  - **__init__(self)**: El constructor de la clase inicializa un grafo vacío. **listaVertices** es un diccionario que almacena los **vértices**, y **numVertices** es un contador para realizar un seguimiento del número de vértices en el grafo.
  - **agregarVertice(self, clave)**: Agrega un nuevo vértice al grafo con la clave dada. Crea una instancia de la clase **Vertice** y la almacena en el diccionario **listaVertices**. Retorna el vértice recién creado.

- **obtenerVertice(self, n)**: Obtiene un vértice por su clave. Si el vértice existe en el grafo, lo devuelve; de lo contrario, devuelve None.
- **__contains__(self, n)**: Permite verificar si un vértice con una clave específica existe en el grafo.
- **agregarArista(self, de, a, ponderacion=0, segunda_ponderacion=0)**: Agrega una arista entre dos vértices con claves desde **de** y hasta **a**. Si los **vértices** no existen, se crean automáticamente. La arista se almacena en el diccionario de vecinos de **de**. Los parámetros **ponderacion** y **segunda_ponderacion** representan las ponderaciones de la arista (peso y precio), siendo esto una tupla.
- **obtenerVertices(self)**: Retorna una lista de las claves de los vértices en el grafo.
- **__iter__(self)**: Permite la iteración a través de los vértices del grafo.
- **crear_grafo_de_rutas_posibles_peso_max(self, inicio, final)**: Crea un nuevo grafo basado en las rutas filtradas que cumplen con el cuello de botella del camino óptimo (calculado con el algoritmo Dijkstra). Retorna el nuevo grafo.
- **crear_grafo_de_rutas_posibles(self, inicio, final, peso)**: Crea un nuevo grafo basado en las rutas filtradas que cumplen con un peso específico. Las rutas son filtradas usando el método **eliminar_rutas_sobrepeso**. Retorna el nuevo grafo.
- **eliminar_rutas_sobrepeso(self, inicio, final, peso_maximo)**: Filtra las rutas que tienen un cuello de botella mayor o igual al peso máximo especificado. Retorna una lista de rutas filtradas.
- **todos_los_cuellos_de_botella_de_rutas_posibles(self, inicio, final)**: Encuentra todos los cuellos de botella de rutas posibles entre el vértice de inicio y el vértice final. Retorna una lista de tuplas llamada “cuellos” y cada tupla contiene (El cuello de botella, "numero de camino", "La ruta:", lista con el id de los vértices de ese camino, "Un str comentando desde que vértice hasta que vértice se encuentra el cuello de botella")
- **encontrador_caminos_posibles(self, inicio, final)**: Encuentra todos los caminos posibles entre el vértice de inicio y el vértice final. Retorna una lista de caminos donde cada camino es una lista de vértices. Este método funciona como una especie de pasador, en el cual utiliza lo que devuelve el método **encontrar_caminos_posibles_str(self, inicio, final),** para pasar La lista de los caminos

(str) a vértices (objetos). Con la finalidad de que sea más sencillo el procesamiento de los caminos.

- **encontrar_caminos_posibles_str(self, inicio, final)**: Función auxiliar para encontrar caminos entre vértices de inicio y final. Retorna una lista de listas, donde cada sublista representa un camino. Este método emplea el algoritmo de Búsqueda en profundidad o también conocida como búsqueda exhaustiva, pero con ciertas modificaciones.

- La clase **Vertice** representa un vértice en un grafo. Cada vértice tiene las siguientes características:

  - **__init__(self)**: El constructor de la clase inicializa un vértice. **id** es un identificador único para el vértice, **conectadoA** es un diccionario que almacena los vértices vecinos como claves y sus ponderaciones como valores en una tupla (ponderación y segunda ponderación), **distancia** es la distancia desde un vértice de inicio hasta este vértice en el contexto de un algoritmo como **Dijkstra**. Inicializado en None, **predecesor** es el predecesor de este vértice en un camino desde un vértice de inicio hasta este vértice. Inicializado en None.
  - **agregarVecino(vecino, ponderacion=0, segunda_ponderacion=0)**: Agrega un vértice vecino al diccionario de vecinos del vértice actual junto con sus ponderaciones (peso y precio), siendo esto una tupla. Los valores predeterminados son 0 para ambas ponderaciones.
  - **__str__()**: Devuelve una representación en cadena del vértice, incluyendo su identificador y los identificadores de sus vecinos.
  - **obtenerConexiones()**: Retorna las claves de los vértices vecinos del vértice.
  - **obtenerId()**: Retorna el identificador del vértice.
  - **obtenerDistancia()**: Retorna la distancia actualmente asignada al vértice (utilizado en algoritmos como Dijkstra).
  - **asignarDistancia(distancia)**: Asigna una distancia específica al vértice.
  - **asignarPredecesor(predecesor)**: Asigna un vértice predecesor al vértice actual.
  - **obtenerPonderacion(vecino)**: Retorna la ponderación (peso) de la arista que conecta el vértice actual con un vértice vecino dado.

- **obtenerSegundaPonderacion(vecino)**: Retorna la segunda ponderación (precio) de la arista que conecta el vértice actual con un vértice vecino dado.

- Dijkstra(unGrafo, inicio): Este método implementa el algoritmo de Dijkstra para encontrar el camino más corto desde un vértice de inicio a todos los demás vértices en un grafo ponderado. Utiliza una cola de prioridad mínima para seleccionar el vértice con la distancia mínima en cada iteración. Aquí está una descripción detallada de su funcionamiento:

  - Inicializa la cola de prioridad mínima, establece todas las distancias a +infinito para todos los vértices en el grafo, excepto el vértice de inicio, cuya distancia se establece en 0.
  - Construye un montículo con todas las distancias y vértices.
  - Mientras la cola de prioridad no esté vacía:
    - Obtén el vértice con la distancia mínima.
    - Explora las conexiones del vértice actual.
    - Calcula la nueva distancia desde el vértice actual al siguiente.
    - Si la nueva distancia es menor que la distancia almacenada en el vértice siguiente, actualiza la distancia del vértice siguiente y establece el vértice actual como su predecesor.

- dijkstra_cuello_botella(unGrafo, inicio, final): Este método modifica el algoritmo de Dijkstra para encontrar el cuello de botella del camino más corto desde un vértice de inicio a un vértice de destino. En lugar de minimizar la distancia, maximiza el cuello de botella (mayor capacidad de carga) a lo largo del camino. La descripción de su funcionamiento es la siguiente:

  - Inicializa la cola de prioridad máxima y establece todas las distancias a 0, excepto el vértice de inicio, cuya distancia se establece en +infinito.
  - Construye un montículo con todas las distancias y vértices.
  - En cada iteración:
    - Obtiene el vértice con la distancia máxima (cuello de botella máximo).
    - Explora las conexiones del vértice actual.

        - Calcula la nueva distancia como el máximo entre la distancia actual y la ponderación del vértice siguiente.
        - Si la nueva distancia es mayor que la distancia almacenada en el vértice siguiente, actualiza la distancia del vértice siguiente y establece el vértice actual como su predecesor.

- **camino_dijkstra(unGrafo, inicio, final)**: Este método, se utiliza para construir el camino desde el vértice de inicio hasta el vértice de destino después de haber ejecutado el algoritmo de **Dijkstra**. Proporciona el camino y la distancia acumulada (precio en este caso) desde el vértice de inicio hasta el vértice de destino. Su funcionamiento es el siguiente:

- Inicializa una lista llamada camino para almacenar los vértices en el camino.
- Obtiene el vértice de destino (final) y lo asigna a **Vfinal**.
- Establece **actual** como el vértice de destino.
- Itera mientras **actual** no sea None.
    - Inserta el vértice actual al principio de la lista camino.
    - Si el vértice inicio es igual al vértice actual, significa que se ha llegado al origen, por lo que el método retorna el camino y la distancia acumulada.
    - Mueve actual al vértice predecesor en el camino más corto.

En resumen, estas clases y métodos son herramientas útiles para trabajar con grafos, encontrar caminos óptimos y gestionar estructuras de datos de colas de prioridad en el contexto de algoritmos como Dijkstra y la búsqueda de rutas