**DESAFIO II**

**Informática II**

**Ethan Salomon Parra Reyes**

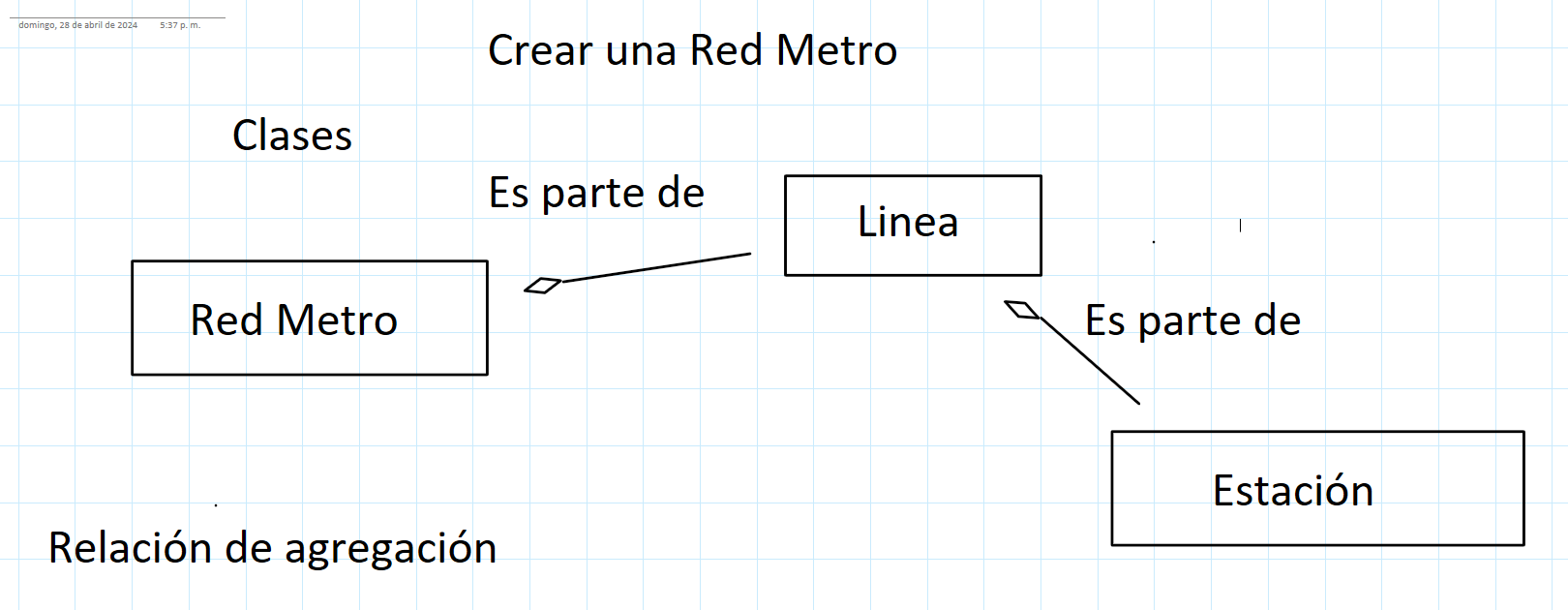
**Juan Felipe García Bonilla**

**Universidad de Antioquia**

**Medellín**

**2024**

**Análisis.**

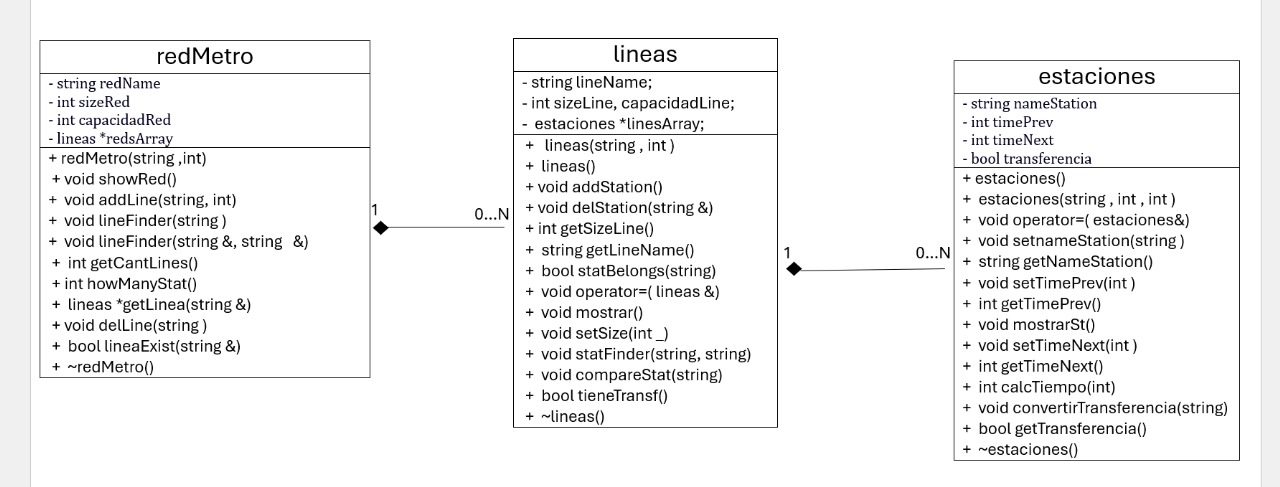


* Se debe implementar un sistema para gestionar el proceso de guardado de la información en las líneas.
* Se debe desarrollar un método que permita el ingreso y retorno de las estaciones con sus respectivos tiempos siguientes o previos con respecto a las estaciones contiguas.
* Se buscara la forma de crear arreglos de tamaño variable, pero sin olvidarse de la eficiencia y el uso de memoria que conlleva este proceso.
* Es importante implementar mecanismos para manejar situaciones inesperadas.
* Hacer métodos para validar que no existan dos líneas ni dos estaciones con el mismo nombre en la red Metro, y de igual forma dos estaciones con el mismo nombre en la misma línea.
* Se deberá crear métodos para procesa la información que es pedida en el DesafioII.
* Se crearan métodos de eliminación de líneas y estaciones, siguiendo las reglas que permiten eliminar una de estas.

C**onsideraciones a tener en cuenta para el desarrollo del problema.**

* Una estación puede pertenecer a varias líneas (si es una estación de transferencia). Los nombres de las estaciones de transferencia se conforman concatenando el nombre de la estación con el nombre de la línea donde se encuentra.
* Una estación sólo puede estar una vez en una línea.
* Una línea sólo puede estar una vez en una red.
* No se pueden crear estaciones independientes a una línea.
* Si una red tiene más de una línea, estas líneas no pueden estar desconectadas.
* Una estación de transferencia puede pertenecer a multiples líneas.
* No se permite que las líneas estén conectadas de tal forma que se creen bucles.
* No podrán haber dos estaciones ni líneas con el mismo nombre.
* Las líneas del metro serán arreglos que contendrán las estaciones y el tiempo a las estaciones contiguas (LineA [re, 2, leA, 5, me, 8, te, 4, ce]).
* Se le preguntara al usuario cual es la estación de transferencia.
* Se duplicara el tamaño de los arreglos de acuerdo a las necesidades, esto con el fin de minimizar el uso de memoria, modificando el tamaño del arreglo cada vez que se ingresa o elimina una nueva estación.
* Cada vez que se cree una nueva estación se le indicara al usuario que si desea crearla en las esquinas de la línea, deberá ingresar un cero ya sea en el tiempo siguiente o previo para identificar la posición de esta nueva estación.

**Diagrama de clases de la solución planteada.**



**Clase estaciones.**

**Constructor predeterminado estaciones()**

Descripción: Constructor predeterminado que inicializa un objeto estaciones con valores por defecto.

Acciones:

Inicializa nameStation como cadena vacía ("").

Establece timePrev y timeNext en 0.

Establece transferencia como false.

**Constructor estaciones(string \_nameStation, int \_timePrev, int \_timeNext)**

Descripción: Constructor que inicializa un objeto estaciones con nombre de estación y tiempos específicos.

Parámetros:

\_nameStation: Nombre de la estación.

\_timePrev: Tiempo a la estación anterior.

\_timeNext: Tiempo a la estación siguiente.

Acciones:

Inicializa nameStation, timePrev y timeNext con los valores proporcionados, establece transferencia como false.

**Sobrecarga del operador de asignación operator=(const estaciones& otraEstacion)**

Descripción: Sobrecarga el operador de asignación para realizar una copia profunda de otro objeto estaciones.

Parámetros:

otraEstacion: Objeto estaciones del cual copiar los atributos.

Acciones:

Copia los atributos (nameStation, timePrev, timeNext y transferencia) de otraEstacion al objeto actual.

**Método mostrarSt()**

Descripción: Muestra los detalles de la estación (nombre, tiempo a la estación anterior y tiempo a la estación siguiente) en la consola.

**Método setTimeNext(int \_timeNext)**

Descripción: Establece el tiempo hacia la estación siguiente.

Parámetros:

\_timeNext: Tiempo a la estación siguiente a establecer.

**Método calcTiempo(int calcTime) const**

Descripción: Calcula el tiempo total a la estación siguiente, sumando el tiempo a la estación siguiente al valor calcTime proporcionado.

Parámetros:

calcTime: Tiempo acumulado hasta el momento.

Retorna: El tiempo total a la estación siguiente.

**Método convertirTransferencia(string lineName)**

Descripción: Marca la estación como una estación de transferencia, agregando el nombre de la línea a la que pertenece al nombre de la estación.

Parámetros:

lineName: Nombre de la línea a agregar al nombre de la estación.

**Método getTransferencia()**

Descripción: Verifica si la estación es una estación de transferencia.

Retorna: true si la estación es una estación de transferencia, false en caso contrario.

Destructor ~estaciones()

Descripción: Destructor que se encarga de liberar la memoria al destruir el objeto estaciones.

**Clase lineas**

**Constructor lineas(string \_lineName, int \_sizeLine)**

Descripción: Este constructor inicializa un objeto lineas con un nombre de línea y un tamaño específico.

Parámetros:

\_lineName: Nombre de la línea.

\_sizeLine: Tamaño inicial de la línea (número de estaciones).

Acciones:

Asigna el nombre de la línea y el tamaño, inicializa el arreglo dinámico linesArray para almacenar las estaciones de la línea y pide al usuario que ingrese el nombre y detalles de cada estación y las agrega al arreglo.

**Constructor predeterminado lineas()**

Descripción: Constructor predeterminado que inicializa un objeto lineas sin parámetros.

Acciones:

Asigna valores predeterminados a lineName, sizeLine y capacidadLine, establece linesArray como nullptr (arreglo vacío).

**Método addStation()**

Descripción: Permite agregar una nueva estación a la línea.

Acciones:

Solicita al usuario que ingrese el nombre de la estación y sus tiempos de conexión con las estaciones anteriores y siguientes, maneja la inserción de la nueva estación en el arreglo linesArray, considerando diferentes escenarios (inicio, final o intermedio del arreglo).

**Método delStation(const string &nameStation)**

Descripción: Elimina una estación de la línea según su nombre.

Parámetros:

nameStation: Nombre de la estación a eliminar.

Acciones:

Busca la estación con el nombre especificado en linesArray y la elimina del arreglo, ajusta el tamaño del arreglo si es necesario.

**Método statBelongs(string nameStat)**

Descripción: Verifica si una estación específica pertenece a la línea.

Parámetros:

nameStat: Nombre de la estación a verificar.

Retorna: true si la estación está en la línea, false en caso contrario.

**Sobrecarga de operador de asignación operator=(const lineas &otraLinea)**

Descripción: Sobrecarga el operador de asignación para realizar una copia profunda de otro objeto lineas.

Parámetros:

otraLinea: Objeto lineas del cual copiar los atributos y el arreglo de estaciones.

Acciones:

Copia los atributos de otraLinea al objeto actual y realiza una copia profunda del arreglo linesArray.

**Método mostrar()**

Descripción: Muestra los detalles de todas las estaciones de la línea.

Acciones:

Itera sobre linesArray y muestra los detalles de cada estación.

**Método setSize(int \_sizeLine)**

Descripción: Establece el tamaño de la línea.

Parámetros:

\_sizeLine: Nuevo tamaño de la línea.

Acciones:

Actualiza sizeLine al valor especificado.

**Método statFinder(string nameStatOrig, string nameStatprev)**

Descripción: Encuentra y calcula el tiempo entre dos estaciones específicas en la línea.

Parámetros:

nameStatOrig: Nombre de la estación de origen.

nameStatprev: Nombre de la estación previa.

Acciones:

Calcula el tiempo entre las dos estaciones especificadas.

**Método compareStat(string NameSt)**

Descripción: Compara una estación específica con todas las estaciones de la línea.

Parámetros:

NameSt: Nombre de la estación a comparar.

Acciones:

Llama a un método de la clase estaciones para convertir una estación en una estación de transferencia si es necesario.

**Método tieneTransf()**

Descripción: Verifica si la línea tiene estaciones de transferencia.

Retorna: true si la línea tiene al menos una estación de transferencia, false en caso contrario.

**Destructor ~lineas()**

Descripción: Libera la memoria utilizada por el arreglo dinámico linesArray al destruir el objeto lineas.

**Clase redMetro**

**Constructor redMetro(string \_redName, int \_sizeRed)**

Descripción:

Este es el constructor de la clase redMetro que inicializa una nueva red de metro con un nombre dado y un tamaño inicial.

Parámetros:

\_redName: Nombre de la red de metro.

\_sizeRed: Tamaño inicial de la red (número de líneas de metro).

Acciones:

Crea un arreglo dinámico redsArray para almacenar las líneas de metro, lee y crea cada línea de metro a partir de la entrada del usuario, verifica si una línea ya existe antes de agregarla al arreglo y por ultimo solicita información sobre estaciones de transferencia para cada línea.

**Método showRed()**

Descripción:

Muestra en consola el nombre de la red de metro y todas las líneas que la componen.

Acciones:

Itera sobre el arreglo redsArray y llama al método mostrar() de cada objeto lineas para mostrar la información de cada línea.

**Método addLine(string lineName, int sizeLine)**

Descripción:

Agrega una nueva línea de metro al arreglo redsArray.

Parámetros:

lineName: Nombre de la nueva línea de metro.

sizeLine: Tamaño (número de estaciones) de la nueva línea.

Acciones:

Aumenta el tamaño de redsArray si es necesario, crea un nuevo objeto lineas con el nombre y tamaño especificados y solicita información sobre estaciones de transferencia para la nueva línea.

**Método lineFinder(string \_lineName)**

Descripción:

Busca una línea de metro por su nombre y realiza una acción sobre ella (por ejemplo, añadir una estación).

Parámetros:

\_lineName: Nombre de la línea de metro a buscar.

Acciones:

Itera sobre redsArray para encontrar la línea con el nombre \_lineName, si encuentra la línea, realiza una acción específica sobre ella (por ejemplo, añadir una estación).

**Método lineFinder(string &\_lineName, string &\_nameStation)**

Descripción:

Busca una línea de metro por su nombre y luego busca una estación dentro de esa línea para realizar una acción (por ejemplo, eliminar la estación).

Parámetros:

\_lineName: Nombre de la línea de metro a buscar.

\_nameStation: Nombre de la estación a buscar dentro de la línea.

Acciones:

Itera sobre redsArray para encontrar la línea con el nombre \_lineName, si encuentra la línea, busca la estación \_nameStation dentro de esa línea y realiza una acción específica sobre ella.

**Método getCantLines()**

Descripción:

Retorna la cantidad de líneas de metro en la red.

Retorno:

Entero que representa la cantidad de líneas de metro (sizeRed).

**Método howManyStat()**

Descripción:

Calcula y retorna la cantidad total de estaciones en todas las líneas de la red.

Retorno:

Entero que representa la cantidad total de estaciones en la red.

**Método getLinea(string &\_lineName)**

Descripción:

Busca y retorna un puntero a la línea de metro con el nombre especificado.

Parámetros:

\_lineName: Nombre de la línea de metro a buscar.

Retorno:

Puntero al objeto lineas que representa la línea de metro encontrada.

**Método delLine(const string \_lineName)**

Descripción:

Elimina una línea de metro del arreglo redsArray.

Parámetros:

\_lineName: Nombre de la línea de metro a eliminar.

Acciones:

Itera sobre redsArray para encontrar la línea con el nombre \_lineName, si encuentra la línea, la elimina del arreglo y ajusta el tamaño y la capacidad del arreglo si es necesario.

**Método lineaExist(string &lineName)**

Descripción:

Verifica si una línea de metro con el nombre especificado ya existe en el arreglo redsArray.

Parámetros:

lineName: Nombre de la línea de metro a verificar.

Retorno:

true si la línea existe, false si no existe.

Destructor ~redMetro()

Descripción:

Libera la memoria asignada dinámicamente para el arreglo redsArray al destruir un objeto redMetro.

Acciones:

Libera la memoria del arreglo redsArray utilizando delete[].

**Algoritmos implementados.**

**Definición de la Clase estaciones.**

**Atributos privados:**

**nameStation:** Una cadena que representa el nombre de la estación.

**timePrev:** Un entero que representa el tiempo a la estación anterior.

**timeNext:** Un entero que representa el tiempo a la estación siguiente.

**transferencia**: Un booleano que indica si la estación es una estación de transferencia.

**Constructores.**

La clase estaciones define dos constructores:

Constructor sin argumentos (estaciones()):

Inicializa los atributos nameStation, timePrev, timeNext y transferencia con valores predeterminados.

Constructor con argumentos (estaciones(string \_nameStation, int \_timePrev, int \_timeNext)):

Permite crear una instancia de estaciones con valores específicos para nameStation, timePrev y timeNext.

**Métodos de Acceso y Modificación.**

**setnameStation(string \_nameStation):** Establece el nombre de la estación.

**getNameStation() const:** Devuelve el nombre de la estación.

**setTimePrev(int \_timePrev):** Establece el tiempo a la estación anterior.

**getTimePrev() const:** Devuelve el tiempo a la estación anterior.

**setTimeNext(int \_timeNext):** Establece el tiempo a la estación siguiente.

**getTimeNext() const:** Devuelve el tiempo a la estación siguiente.

**Otros Métodos.**

**mostrarSt():** Muestra en la consola el nombre de la estación, el tiempo a la estación anterior y el tiempo a la estación siguiente.

**calcTiempo(int calcTime) const:** Calcula el tiempo acumulado sumando el tiempo a la estación siguiente al valor pasado como parámetro calcTime.

**convertirTransferencia(string lineName):** Convierte la estación en una estación de transferencia agregando el nombre de la línea lineName al nombre de la estación.

**getTransferencia():** Devuelve true si la estación es una estación de transferencia, de lo contrario devuelve false.

**Destructor.**

El destructor ~estaciones() es responsable de liberar cualquier recurso asignado dinámicamente (aunque en este caso no realiza ninguna acción específica).

**Uso del Operador de Asignación (operator=).**

El método operator= permite asignar una estación a otra, copiando los valores de sus atributos.

**Definición de la Clase lineas.**

**La clase lineas tiene los siguientes atributos privados:**

**lineName:** Una cadena que representa el nombre de la línea.

**sizeLine:** Un entero que indica el número actual de estaciones en la línea.

**capacidadLine:** Un entero que indica la capacidad actual del arreglo dinámico linesArray.

**linesArray:** Un puntero a objetos de tipo estaciones, que representa un arreglo dinámico de estaciones.

**Constructores.**

Constructor con parámetros (lineas(string \_lineName, int \_sizeLine)):

Este constructor inicializa una línea con un nombre (\_lineName) y un número inicial de estaciones (\_sizeLine).

Crea un arreglo dinámico linesArray para almacenar las estaciones, inicializando su capacidad inicial (capacidadLine) y asignando valores a cada estación según las entradas del usuario.

Constructor sin parámetros (lineas()):

Este constructor inicializa una línea sin ningún nombre ni estaciones. La capacidad inicial se establece en un valor predeterminado.

**Métodos.**

**addStation():** Permite agregar una nueva estación a la línea, solicitando al usuario el nombre y tiempos de conexión de la estación.

Solicita al usuario el nombre de la nueva estación y los tiempos de conexión con la estación anterior y siguiente.

Verifica si la estación ya existe en la línea usando statBelongs().

Si no existe, decide dónde insertar la nueva estación en el arreglo linesArray según los tiempos de conexión (TimeP y TimeN).

Si el tamaño actual de la línea supera la capacidad actual del arreglo, duplica la capacidad (capacidadLine) y crea un nuevo arreglo dinámico con la capacidad actualizada.

Inserta la nueva estación en el lugar correcto del arreglo linesArray.

Actualiza el tamaño de la línea (sizeLine).

**delStation(const string &nameStation):** Elimina una estación específica de la línea según su nombre.

Recorre el arreglo linesArray para encontrar la estación con el nombre especificado.

Si encuentra la estación, la elimina moviendo las estaciones restantes hacia adelante en el arreglo.

Verifica si es necesario reducir la capacidad del arreglo linesArray.

**getSizeLine():** Devuelve el tamaño actual de la línea (número de estaciones).

**getLineName() const:** Devuelve el nombre de la línea.

**statBelongs(string nameStat):** Verifica si una estación con un nombre específico ya pertenece a la línea.

**operator=:** Sobrecarga el operador de asignación para copiar una línea existente en otra línea.

**mostrar():** Muestra en consola el nombre de la línea y detalles de cada estación.

**setSize(int \_sizeLine):** Establece el tamaño de la línea.

**statFinder(string nameStatOrig, string nameStatprev):** Encuentra el tiempo acumulado entre dos estaciones específicas.

**compareStat(string NameSt):** Convierte una estación en una estación de transferencia dentro de la línea.

**tieneTransf():** Verifica si la línea tiene al menos una estación de transferencia.

**Otros Métodos**

**statFinder():** Encuentra el tiempo acumulado entre dos estaciones específicas (nameStatOrig y nameStatprev).

**compareStat():** Convierte una estación en una estación de transferencia si es necesario.

**tieneTransf():** Verifica si la línea tiene al menos una estación de transferencia.

**Destructor.**

El destructor ~lineas() libera la memoria asignada dinámicamente para el arreglo linesArray.

**Definición de la Clase redMetro**

**La clase redMetro tiene los siguientes atributos privados:**

**redName:** Una cadena que representa el nombre de la red de metro.

**sizeRed:** Un entero que indica el número actual de líneas en la red.

**capacidadRed:** Un entero que indica la capacidad actual del arreglo dinámico redsArray.

**redsArray:** Un puntero a objetos de tipo lineas, que representa un arreglo dinámico de líneas de metro.

**Constructor.**

El constructor redMetro(string \_redName, int \_sizeRed) inicializa una red de metro con un nombre (\_redName) y un número inicial de líneas (\_sizeRed). En este constructor:

Se solicita al usuario información sobre cada línea, como el nombre y la cantidad inicial de estaciones.

Se crea un nuevo arreglo dinámico redsArray para almacenar las líneas de metro.

Se verifica si una línea con el mismo nombre ya existe en la red utilizando el método lineaExist().

Si no existe, se crea una nueva instancia de lineas y se agrega al arreglo redsArray.

Se llama al método compareStat() para cada línea existente, lo que permite convertir una estación en una estación de transferencia si es necesario.

**Métodos.**

**showRed():** Muestra en consola el nombre de la red de metro y detalles de cada línea.

**addLine(string lineName, int sizeLine):** Agrega una nueva línea a la red de metro con un nombre (lineName) y una cantidad inicial de estaciones (sizeLine).

Incrementa el tamaño de la red de metro (sizeRed) para agregar una nueva línea.

Verifica si la capacidad actual del arreglo (redsArray) es suficiente para almacenar la nueva línea.

Si es necesario, duplica la capacidad del arreglo y crea un nuevo arreglo dinámico para almacenar las líneas existentes.

Agrega la nueva línea al final del arreglo redsArray.

Llama al método compareStat() para convertir una estación en una estación de transferencia si es necesario.

**lineFinder(string \_lineName):** Busca una línea específica en la red de metro y permite agregar una estación a esa línea llamando al método addStation() de la clase lineas.

**lineFinder(string &\_lineName, string &\_nameStation):** Busca una línea y una estación específicas en la red de metro y permite eliminar esa estación de esa línea llamando al método delStation() de la clase lineas.

**getCantLines():** Devuelve la cantidad actual de líneas en la red de metro.

**howManyStat():** Devuelve el total de estaciones en todas las líneas de la red de metro.

getLinea(string &\_lineName) cons**t:** Busca y devuelve un puntero a una línea específica en la red de metro.

**delLine(const string \_lineName):** Elimina una línea específica de la red de metro.

Recorre el arreglo redsArray para encontrar la línea con el nombre especificado.

Si encuentra la línea, la elimina moviendo las líneas restantes hacia adelante en el arreglo.

Verifica si el tamaño actual de la red de metro es una potencia de dos después de eliminar la línea.

Si es una potencia de dos, reduce la capacidad del arreglo redsArray.

**lineaExist(string &lineName):** Verifica si una línea con un nombre específico ya existe en la red de metro.

**Otros Métodos.**

**lineFinder():** Permite encontrar una línea específica en la red de metro y realizar operaciones relacionadas.

**howManyStat():** Calcula el total de estaciones en todas las líneas de la red de metro.

**getLinea():** Permite obtener un puntero a una línea específica en la red de metro.

**lineaExist():** Verifica si una línea con un nombre específico ya existe en la red de metro.

**Destructor.**

El destructor ~redMetro() libera la memoria asignada dinámicamente para el arreglo redsArray.

**Problemas de desarrollo que afrontó.**

1. Desafíos en el manejo de excepciones y creación de contingencias.
2. Desafíos en la elaboración de un programa eficiente y modular.
3. Dificultades para establecer condiciones que optimicen la eficiencia del programa.
4. Dificultades con un buen uso de arreglos dinámicos.
5. Dificultades con el tiempo para poder cumplir con el desafioII.

**Evolución de la solución y consideraciones para tener en cuenta en la implementación.**

* Para hacer un buen uso del programa y sus opciones, se le pide al usuario crear una red Metro desde el principio.
* Se crearon arreglos dinámicos que contendrán los objetos ya sea de estaciones o de líneas.
* Se diseñaron he implementaron los métodos solicitados en el DesafioII, tales como añadir una línea a la red metro o una estación a una línea, estos métodos se desarrollaron haciendo uso de los arreglos dinámicos, los cuales almacenaran los objetos, además en estos métodos, se le pregunta al usuario cual será la estación de transferencia.
* Se creó un método con el fin de concatenar el nombre de la línea con el nombre de las estaciones para así identificar las estaciones de transferencia.
* Para hacer un uso más eficiente de la memoria, se determinó que los arreglos no iban a cambiar constantemente de tamaño, sino, que estos aumentarían su tamaño en múltiplos de dos.
* Para saber cuántas estaciones hay en una línea y cuantas líneas hay en una red Metro, se implementaron métodos que llamaran los atributos que denotan la cantidad de cada una de estas.
* Para comprobar que si se pueda eliminar una línea o una estación se implementaron métodos con el fin de comprobar que las líneas no tuvieran estación de transferencia y que las estaciones no fueran de transferencia para permitir su eliminación.
* Ya que el tiempo de llegada entre sí, de dos estaciones contiguas es el mismo, se denoto que para calcular este tiempo de llegada de una a estación a otra solo se necesitaría calcular la sumatoria de uno de los tiempos, fuese el tiempo siguiente o anterior.
* Para crear un programa más amigable con el usuario, se diseñó una interfaz.
* Se ha aplicado el manejo de excepciones para evitar que el usuario ingrese valores fuera de los parámetros requeridos por el programa.
* Se ha establecido métodos para comprobar que los nombres de las estaciones y líneas ingresados por el usuario existan en la red, de igual forma estos métodos se implementar con el fin de impedir que hallan estaciones o líneas con el mismo nombre.

**Red Metro.**

Atributos.

* nameRed.

Métodos.

**Línea.**

Atributos.

* nameLine.
* arrayStation.

Métodos.

* constructorLine.
* getarrayStation.
* setarratStation.
* createLine.
* deleteLine.
* addStation\_array.
* deleteStation\_array
* calcularTime.
* contStation\_Line
* stationInLine

**Estación.**

Atributos.

* nameStation.
* timePrev.
* timeNext.

Métodos.

* constructorStation.
* getStation.
* setStation.
* createStation.
* deleteStation.