INFORME DESAFIO 2

Autores

John Axel Ortega Rivera

Victor David Muñoz Ramirez

Profesores

Augusto salazar

Anibal

Universidad de Antioquia

Carrera de ingeniería electrónica

Medellín, Antioquia

24 abril de 2024

Índice:

Problemática:

- **Breve descripción del problema que se va a resolver.**

Se requiere desarrollar un simulador de una red de Metro que permita modelar algunas características de su funcionamiento. Una red de Metro está compuesta por líneas, y cada línea tiene una secuencia de estaciones. Algunas estaciones pueden ser estaciones de transferencia, perteneciendo a múltiples líneas. El simulador debe permitir agregar/eliminar estaciones y líneas, así como realizar operaciones como calcular el tiempo de llegada de un tren entre dos estaciones de la misma línea. El diseño debe seguir los principios de la Programación Orientada a Objetos (POO) y utilizar estructuras de datos eficientes, como arreglos dinámicos, sin el uso de contenedores de la biblioteca estándar de C++.

- **Identificación de requisitos funcionales y no funcionales.**

Requisitos funcionales:

Funciones y métodos → Apartado en el diseño de la solución.

Entradas y salidas → Estaciones y líneas. Red de metro como salida

Excepciones y verificaciones → Estaciones, estaciones de transferencia y líneas.

Requisitos no funcionales:

Clases, arreglo dinámico → capacidad de trabajo: El programa debe ser capaz de gestionar eficientemente la memoria y evitar fugas de memoria. → **arquitectura apropiada:** El programa debe contar con una arquitectura acorde para la resolución del problema, mostrando así el resultado más eficiente y estructurado.

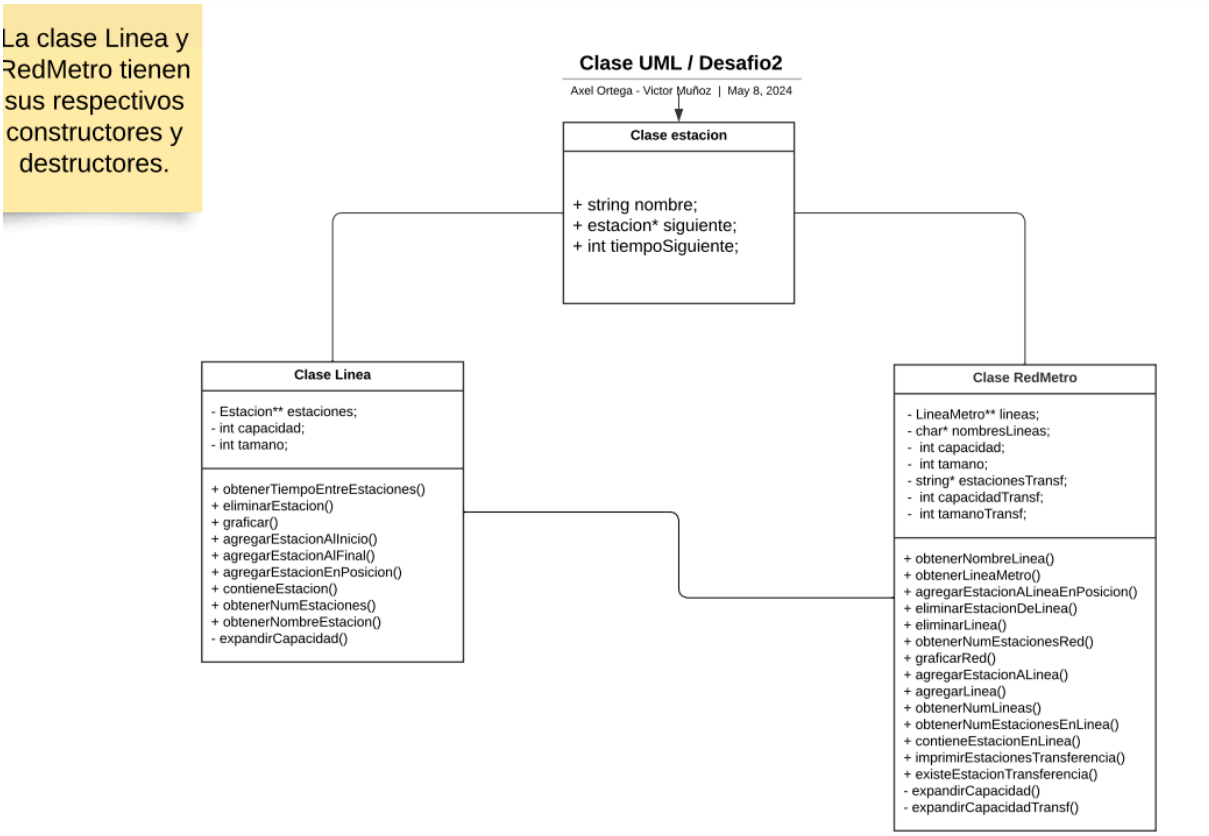
Análisis del problema:

- **Identificación de las entradas y salidas del programa.**

Entradas: Estaciones y líneas, y sus atributos, para las estaciones el nombre, los tiempos anteriores y siguientes de las estaciones allegadas; y las líneas sólo su nombre, y si hay más de dos líneas, cuál es la estación de transferencia que la conecta con otra línea.

Salidas: Red de metro ó los apartados que están en el menú.

Diagrama de clases:



Diseño de la solución:

- **Explicación de la arquitectura general del programa.**

Variables globales: std::string nombreEstacion , std::string nombreEstacionFin, int posicion, int tiempoAnterior, int tiempoSiguiente, int numEstaciones, char nombreLinea, char opcion;

Programa Principal: El programa principal consta del menú que se mostrará en el monitor repetidamente hasta que el usuario indique el momento de salida; el menú como tal

funcionara con un do while y con switch, donde cada una de las opciones hará las llamadas necesarias para cumplir su objetivo, en base a esto las opciones del programa serían las siguientes:

A. Agregar una estación a una línea.

B. Eliminar una estación de una línea.

C. Saber cuantas lineas tiene la red Metro.

D. Saber cuántas estaciones tiene una línea dada.

E. Saber si una estación dada pertenece a una línea específica.

F. Agregar una línea a la red Metro.

G. Eliminar una línea de la red Metro.

H. Saber cuántas estaciones tiene la red Metro.

I. Graficar la red Metro.

J. Saber el tiempo de llegada entre estaciones de una misma linea.

K. Mostrar estaciones de transferencia.

X. Salir.

- **Métodos:**

Estación:

- **Estacion(const std::string& nombre, int tiempoSiguiente = 0):** Constructor de la clase Estación. Crea una nueva instancia de Estación con un nombre y tiempo de viaje a la siguiente estación. El tiempo de viaje a la siguiente estación es opcional y por defecto es 0.

LineaMetro:

1. **obtenerNombreEstacion(int indice):** Devuelve el nombre de la estación en la posición dada. Si el índice está fuera de rango, retorna una cadena vacía. Permite acceder al nombre de una estación específica en la línea.
2. **eliminarEstacion(const std::string& nombre):** Elimina la estación con el nombre dado de la línea. Ajusta los tiempos de viaje entre estaciones según la posición

eliminada. Retorna verdadero si la eliminación fue exitosa, falso si la estación no existe.

3. **graficar():** Imprime una representación gráfica de la línea de metro. Muestra los nombres de las estaciones y los tiempos de viaje entre ellas. Proporciona una visualización clara de la estructura de la línea.
4. **expandirCapacidad():** Función privada que expande la capacidad del arreglo de estaciones. Se llama automáticamente cuando se agrega una nueva estación y no hay espacio. Garantiza que haya suficiente memoria para almacenar nuevas estaciones.
5. **agregarEstacionAlInicio(const std::string& nombre, int tiempoSiguiente):** Agrega una nueva estación al inicio de la línea. Requiere el nombre de la estación y el tiempo de viaje a la siguiente. Ajusta los tiempos de viaje de las estaciones existentes.
6. **agregarEstacionAlFinal(const std::string& nombre, int tiempoAnterior):** Agrega una nueva estación al final de la línea. Requiere el nombre de la estación y el tiempo de viaje desde la anterior. Ajusta el tiempo de viaje de la estación previa.
7. **agregarEstacionEnPosicion(const std::string& nombre, int posicion, int tiempoAnterior, int tiempoSiguiente):** Agrega una nueva estación en una posición específica de la línea. Requiere el nombre, la posición, y los tiempos de viaje anterior y siguiente. Ajusta los tiempos de viaje de las estaciones adyacentes.
8. **contieneEstacion(const std::string& nombre):** Verifica si una estación con el nombre dado existe en la línea. Retorna verdadero si la estación está presente, falso en caso contrario. Útil para comprobar la existencia de una estación antes de realizar operaciones.
9. **obtenerNumEstaciones():** Devuelve la cantidad total de estaciones en la línea. Proporciona información sobre el tamaño de la línea de metro. Es útil para iteraciones o cálculos relacionados con la línea.
10. **obtenerTiempoEntreEstaciones(const std::string& nombreEstacionInicio, const std::string& nombreEstacionFin):** Calcula el tiempo total de viaje entre dos estaciones dadas. Verifica si las estaciones existen y suma los tiempos de viaje intermedios. Retorna el tiempo total o -1 si alguna estación no existe.

RedMetro:

1. **existeEstacionTransferencia(const std::string& nombreEstacion):** Verifica si una estación con el nombre dado es una estación de transferencia. Recorre el arreglo de estaciones de transferencia y compara los nombres. Retorna verdadero si la estación es de transferencia, falso en caso contrario.
2. **expandirCapacidadTransf():** Función privada que expande la capacidad del arreglo de estaciones de transferencia. Se llama automáticamente cuando se agrega una nueva estación de transferencia y no hay espacio. Garantiza que haya suficiente memoria para almacenar nuevas estaciones de transferencia.
3. **~RedMetro():** Destructor de la clase RedMetro. Libera la memoria dinámica utilizada por los arreglos de líneas, nombres de líneas y estaciones de transferencia. Asegura que no haya fugas de memoria al eliminar la instancia de RedMetro.

4. **obtenerLineaMetro(int indice):** Devuelve un puntero a la instancia de LineaMetro en la posición dada. Si el índice está fuera de rango, el comportamiento es indefinido. Proporciona acceso directo a una línea de metro específica en la red.
5. **agregarEstacionALineaEnPosicion(const char& nombreLinea, const std::string& nombreEstacion, int posicion, const int& tiempoAnterior, const int& tiempoSiguiente):** Agrega una nueva estación a una línea existente en una posición específica. Requiere el nombre de la línea, nombre de la estación, posición, y tiempos de viaje anterior y siguiente. Si la línea no existe, muestra un mensaje de error.
6. **eliminarEstacionDeLinea(const char& nombreLinea, const std::string& nombreEstacion):** Elimina una estación de una línea existente en la red. Requiere el nombre de la línea y el nombre de la estación. Si la estación es una estación de transferencia, no se puede eliminar y muestra un mensaje de error. Si la línea no existe, muestra un mensaje de error.
7. **eliminarLinea(const char& nombreLinea):** Elimina una línea completa de la red de metro. Requiere el nombre de la línea a eliminar. Si la línea contiene una estación de transferencia, no se puede eliminar y muestra un mensaje de error. Si la línea no existe, muestra un mensaje de error.
8. **obtenerNumEstacionesRed():** Calcula y devuelve la cantidad total de estaciones en toda la red de metro. Suma las estaciones de cada línea, pero resta la cantidad de líneas y suma 1 para evitar contar repeticiones.
9. **graficarRed():** Imprime una representación gráfica de toda la red de metro. Itera sobre cada línea y llama al método graficar() de cada una. Proporciona una visualización clara y completa de la estructura de la red.
10. **agregarLinea(const char& nombreLinea):** Agrega una nueva línea a la red de metro. Requiere el nombre (carácter) de la nueva línea. Si la línea ya existe, muestra un mensaje de error. Si es la primera línea, solo solicita el nombre de la primera estación. Si no es la primera línea, solicita la línea y estación de transferencia.
11. **obtenerNumEstacionesEnLinea(const char& nombreLinea):** Devuelve la cantidad de estaciones en una línea específica. Requiere el nombre de la línea. Si la línea no existe, retorna -1.

- **Descripción de las estructuras de datos utilizadas.**

Arreglos dinámicos→ Son estructuras de datos flexibles cuyo tamaño puede cambiar durante la ejecución del programa. A diferencia de los arreglos estáticos, cuyo tamaño se define en tiempo de compilación, los arreglos dinámicos se asignan en tiempo de ejecución utilizando punteros. Esto permite la gestión eficiente de la memoria, ya que se asigna solo la cantidad necesaria de memoria en el momento necesario. Los arreglos dinámicos son útiles

cuando el tamaño del conjunto de datos no se conoce de antemano o puede cambiar durante la ejecución del programa.

Clases → Son estructuras que permiten agrupar datos y funciones relacionadas en un solo objeto. Utilizadas en programación orientada a objetos (POO), ofrecen encapsulación para controlar el acceso a los datos y abstracción para modelar entidades del mundo real. Los constructores y destructores son métodos especiales que se utilizan para inicializar y destruir objetos, respectivamente. Los modificadores de acceso, como public, private y protected, controlan la visibilidad de los miembros de la clase, asegurando una encapsulación efectiva.

- **Algoritmos empleados para resolver el problema.**

Clases, arreglos dinámicos, operaciones con punteros, operaciones aritméticas.

- **Problemas de desarrollo:**

El problema más grande al que nos enfrentamos durante el desarrollo del programa fue establecer y analizar la arquitectura del programa más conveniente para la realización del desafío, a su vez saber que atributos y de qué manera iba a funcionar el programa fue un reto bastante grande. Esto debido a que como fuimos implementando poco a poco cada lógica habian métodos que constantemente estaban en cambio, ideas que se nos iban surgiendo para resolver el programa de una manera más eficiente también fue un inconveniente a la hora de analizar de manera detalla y especifica el funcionamiento, todo esto está documentado en la parte de los anexos y más específicamente en el bloc de notas que esta el dia a dia del análisis que íbamos realizando.

A su vez nos gustaría mencionar y darle importancia a los arreglos dinámicos, debido a que estos fueron el primer problema al que nos enfrentamos en el momento de hacer el programa, ya que básicamente nuestro programa gira en base al funcionamiento de dichos arreglos dinámicos.

Por último antes de terminar este apartado, cabe mencionar que nos gustaron los retos afrontados en este parcial pese a las dificultades ya mencionadas, y inclusive algunas no mencionadas que son más triviales, como lo sería los métodos más “sencillos” como el de verificar si una estación estaba en una línea dada, este parcial nos ayudó a mejorar mucho el análisis de las problemáticas, y un cambio que vimos con respecto al anterior parcial fue la comunicación y la sinergia, debido a que en esta ocasión no tuvimos inconvenientes con nombres de variables, métodos o cualquier otro tipo de datos, ya que básicamente estuvimos más atentos de esos pequeños detalles y trabajamos en equipo constantemente.

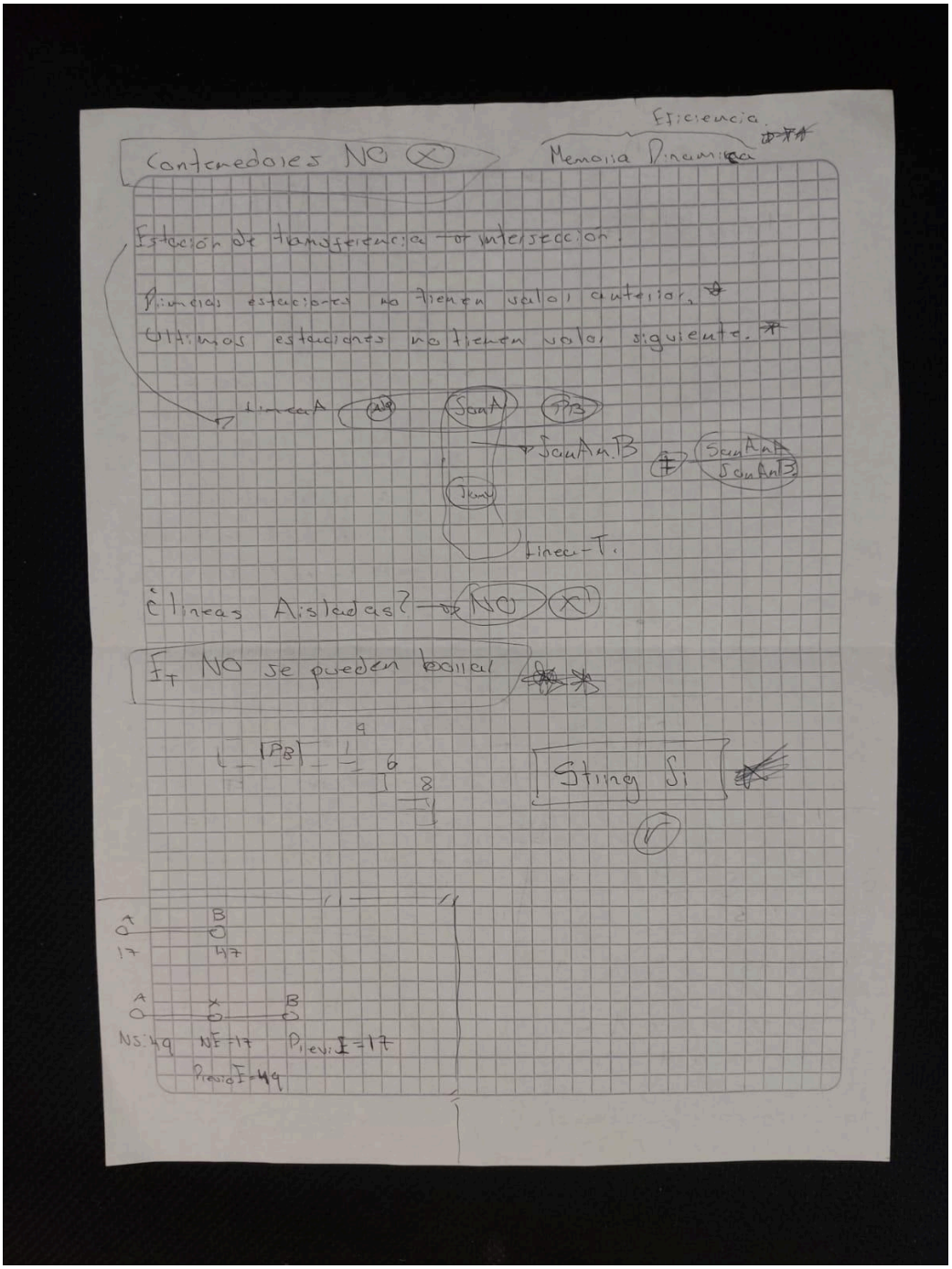
- **Evolución de la solución y consideraciones**

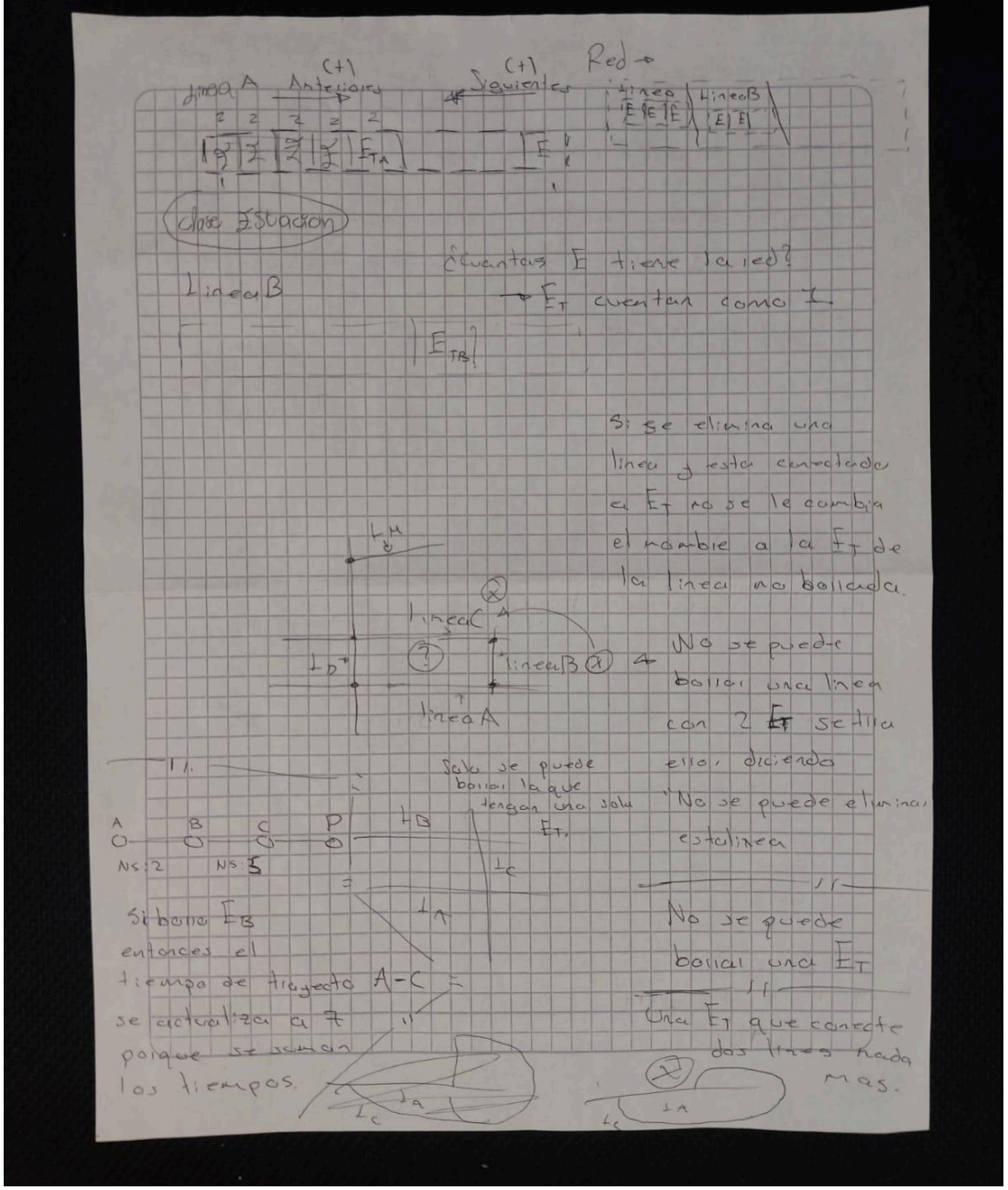
Durante el parcial y la evolución de la solución se tuvo en cuenta sobretodo el cómo establecer la estructura del programa de la manera más eficiente posible como lo hemos mencionado anteriormente, a su vez como tener un orden adecuado y que estuvieran lo mas organizadas posibles, a su vez consideraciones iniciales como se ven en los anexos se tuvieron en cuenta sobre todo al principio y durante el desarrollo del programa, en otras palabras, ir realizando el desafío por versiones fue una muy buena idea, ya que básicamente teníamos una base en donde trabajar durante todo el desarrollo, establecer que primero iba a ir una versión con solo algunos métodos más fundamentales que otros, después agregar la lógica de la posición, siguiente a eso la lógica del tiempo y por último la lógica de las estaciones de transferencia nos facilitó mucho el trabajo ya que a esa base que creamos al principio la íbamos estructurando mas y mas al pasar del tiempo, así para tener al final el programa totalmente realizado; esto se ve evidenciado en el video y en la documentación aquí presente. Por último cabe mencionar que algunos atributos inclusive en el desarrollo tuvieron cambios debido a que nos dimos cuenta de que necesitábamos reestructurarlos para poder aplicar las lógicas correspondientes, a su vez en la historia de los documentos se ve la evolución del diagrama de clases y como varios conceptos cambiaron durante el desarrollo del programa.

Anexos

En el siguiente link se muestran las notas/análisis/planteamientos del proyecto que se van realizando a lo largo del desarrollo del parcial.

https://udeaeduco-my.sharepoint.com/:o:/g/personal/axel_ortega_udea_edu_co/Ev9QS07zHINPkdne45qWs3kBIX0iEb1o826H8UJSmDoc0g?e=S9bwc0




Línea A
(+1) Anteriores
(+1) Siguientes
Red
Línea B
clase ESTACION
¿Cuantas E tiene la red?
→ ET cuentan como 1.
Línea B
ETB
Si se elimina una linea y esta conectada a ET no se le cambia el nombre a la ET de la linea no bolleada.
LM
Línea C
Línea B
Línea A
No se puede bolar una linea con 2 ET se lia ello, diciendo "No se puede eliminar esta linea"
A
B
C
D
NS: 2
NS: 5
Solo se puede bolar la que tengan una sola ET.
Si bolla ETB entonces el tiempo de trayecto A-C = se actualiza a ∞ porque se anulan los tiempos.
No se puede bolar una ET
Una ET que conecte dos lineas nada mas.