**Simulador de Red de Metro**

**Alvaro Fernando Gomez Caicedo**

**Juan David Jimenez Marin**

**A. Análisis del problema y consideraciones para la alternativa de solución propuesta**

**Análisis del problema**

El problema planteado consiste en el desarrollo de un simulador que permita modelar algunas características del funcionamiento de un sistema de transporte tipo Metro. El objetivo principal es representar la estructura de una red de Metro, la cual está compuesta por líneas que conectan diferentes estaciones. Algunas de estas estaciones pueden ser puntos de transferencia entre líneas.

El análisis del problema revela una serie de entidades y relaciones que deben ser adecuadamente representadas:

* Red de Metro: Es el elemento principal que engloba todas las líneas y estaciones.
* Línea: Representa una ruta específica del Metro, identificada por un nombre y un tipo de transporte (tren o tranvía). Cada línea está compuesta por una secuencia de estaciones.
* Estación: Es un punto en la red donde se puede abordar o descender del transporte. Cada estación pertenece a una o más líneas (en caso de ser una estación de transferencia) y tiene un nombre único.
* Estación de Transferencia: Es una estación especial que permite la conexión entre dos o más líneas diferentes.

Además, se deben considerar ciertas reglas y restricciones, como que una estación no puede aparecer más de una vez en una línea, una línea no puede pertenecer a más de una red, y si una red tiene múltiples líneas, estas deben estar conectadas.

**Consideraciones para la alternativa de solución**

La solución propuesta se basará en el paradigma de Programación Orientada a Objetos (POO) en C++, lo cual permitirá modelar adecuadamente las entidades y relaciones identificadas en el análisis.

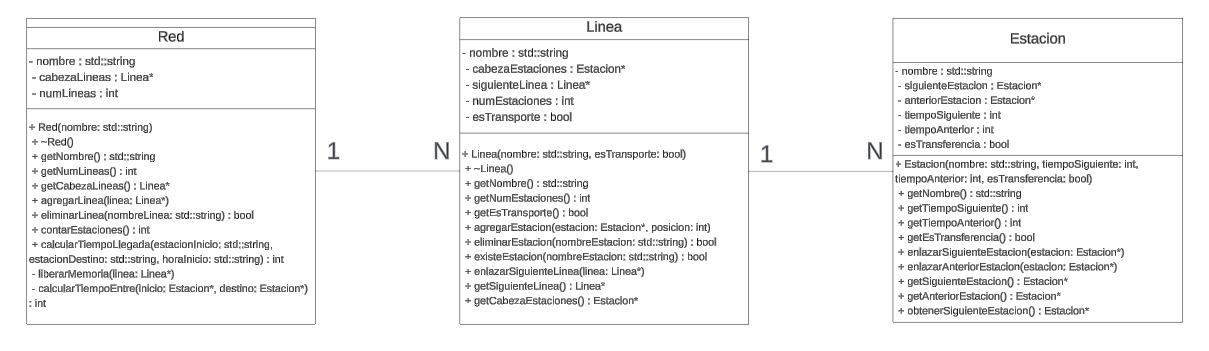
Se plantea la creación de tres clases principales: Red, Línea y Estación. Cada una de estas clases tendrá sus respectivos atributos y métodos para representar sus características y comportamientos.

La clase Red será la clase principal que contendrá y gestionará las líneas y estaciones. Cada instancia de la clase Línea tendrá una secuencia de estaciones asociadas. La clase Estación representará una estación individual, y en caso de ser una estación de transferencia, estará asociada a múltiples líneas.

Se implementarán funciones para agregar, eliminar y consultar información sobre estaciones y líneas dentro de la red. Además, se desarrollará un subprograma específico para simular el cálculo del tiempo de llegada entre dos estaciones dentro de una misma línea.

La solución se enfocará en cumplir con los requisitos mínimos establecidos, prestando especial atención a la eficiencia en el diseño de estructuras de datos y la implementación de los algoritmos necesarios.

**B)Diagrama de clases de la solución planteada. Adicionalmente, describa en alto nivel la lógica de las tareas que usted definió para aquellos subprogramas cuya solución no sea trivial.**

****

**Nota:**Si no es legible el diagrama UML simplificado visite esta pagina <https://lucid.app/lucidchart/4f9568e8-4e29-47d3-9e10-863ec6555ef2/edit?viewport_loc=-365%2C301%2C3330%2C1509%2C0_0&invitationId=inv_b2c0c585-b957-4dc8-9922-a2e5757a9598>

allí encontrará el diagrama de forma clara. O también se recomienda hacer zoom

1. **Agregar una línea de metro:**

-Solicita al usuario ingresar el nombre de la línea.

-Crea un objeto linea con el nombre proporcionado.

-Agrega la línea a la red de metro.

1. **Eliminar una línea de metro:**

-Solicita al usuario el nombre de la línea a eliminar.

-Busca la línea en la red de metro.

-Verifica si la línea tiene una estación de transferencia.

-Si no tiene transferencia, elimina la línea de la red.

1. **Agregar una estación a una línea de metro:**

-Solicita al usuario ingresar el nombre de la estación, tiempos de llegada y salida, si es de transferencia, y la línea a la que pertenece.

-Busca la línea a la que pertenece la estación.

-Verifica la validez de la posición donde se agregará la estación.

-Agrega la estación a la línea en la posición especificada.

1. **Eliminar una estación de una línea de metro:**

-Solicita al usuario el nombre de la estación y la línea a la que pertenece.

-Busca la línea y la estación en la red de metro.

-Elimina la estación de la línea si existe.

1. **Verificar la existencia de una estación en una línea de metro:**

-Solicita al usuario el nombre de la estación y la línea.

-Busca la línea y la estación en la red de metro.

-Informa al usuario si la estación existe en la línea o no.

1. **Contar las estaciones en una línea de metro:**

-Solicita al usuario el nombre de la línea.

-Busca la línea en la red de metro.

-Cuenta y muestra la cantidad de estaciones en esa línea.

1. **Contar las estaciones en toda la red de metro:**

-Cuenta y muestra la cantidad total de estaciones en todas las líneas de la red de metro.

1. **Calcular tiempo de llegada entre dos estaciones:**

-Solicita al usuario el nombre de la estación de inicio, la estación de destino y la hora de inicio.

-Busca las estaciones y la línea a la que pertenecen en la red de metro.

-Calcula el tiempo estimado de llegada entre las dos estaciones, considerando la hora de inicio proporcionada.

**D)Problemas de desarrollo que afrontó.**

1. **Diseño de la estructura de clases:**

-Tuvimos que pensar detenidamente en cómo organizar las clases para representar de manera efectiva la red de metro, las líneas, las estaciones y sus relaciones. Queríamos una estructura que fuera fácil de entender y modificar si era necesario en el futuro.

1. **Gestión de memoria dinámica:**

-Uno de los desafíos fue asegurarnos de asignar y liberar correctamente la memoria dinámica para evitar problemas como fugas de memoria o accesos a memoria no válidos. Necesitábamos ser muy cuidadosos al trabajar con punteros y asegurarnos de liberar la memoria cuando ya no fuera necesaria.

1. **Interfaz de usuario:**

-Creamos una interfaz de usuario intuitiva para que los usuarios pudieran interactuar fácilmente con el programa. Esto implicó diseñar un menú claro con opciones comprensibles, validar las entradas del usuario y proporcionar retroalimentación útil durante la interacción.

1. **Lógica de operaciones:**

-Implementamos la lógica para agregar y eliminar líneas y estaciones, verificar la existencia de estaciones, contar estaciones en líneas y en la red, y calcular tiempos de llegada. Tuvimos que considerar varios escenarios posibles y asegurarnos de manejarlos correctamente.

1. **Manejo de errores y excepciones:**

-Nos enfrentamos al desafío de manejar errores y excepciones de manera adecuada para que el programa no se bloqueara ni produjera resultados incorrectos debido a entradas inválidas del usuario o acciones no permitidas.

1. **Pruebas y depuración:**

-Realizamos pruebas exhaustivas del programa para identificar y corregir cualquier error o comportamiento inesperado. La depuración fue una parte crucial del proceso para garantizar que el programa funcionara como se esperaba en diferentes situaciones.

1. **Optimización de rendimiento:**

-Buscamos optimizar el rendimiento del programa mediante la mejora de algoritmos y estructuras de datos. Queríamos que el programa fuera eficiente, especialmente al trabajar con grandes cantidades de datos.

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

**Evolución de la solución:**

Diseño inicial: Comenzamos con un diseño básico que incluía las clases Red, Linea y Estacion Definimos las relaciones entre estas clases para representar la estructura de una red de metro.

-Iteraciones y mejoras: A medida que avanzamos, iteramos sobre el diseño, agregando funcionalidades como la capacidad de agregar y eliminar líneas y estaciones, verificar la existencia de estaciones, calcular tiempos de llegada, entre otros. También mejoramos la gestión de memoria y la lógica de operaciones.

-Refinamiento de la interfaz de usuario: Trabajamos en mejorar la interfaz de usuario para que fuera más intuitiva y fácil de usar. Esto implicó agregar un menú claro con opciones comprensibles y mensajes de retroalimentación para guiar al usuario durante la interacción.

-Pruebas y correcciones: Realizamos pruebas exhaustivas para identificar errores y comportamientos inesperados, corrigiendo problemas y optimizando el rendimiento del programa.

**Consideraciones para la implementación:**

-Diseño modular y escalable: Es crucial mantener un diseño modular y escalable para facilitar futuras actualizaciones y extensiones del programa. Las clases y funciones deben estar bien organizadas y separadas por responsabilidades.

-Gestión de memoria robusta: Dado que el programa trabaja con memoria dinámica, es importante garantizar una gestión de memoria robusta para evitar fugas de memoria y accesos a memoria no válidos. Se deben liberar los recursos adecuadamente cuando ya no sean necesarios.

-Validación de entradas: Debe haber una validación estricta de las entradas del usuario para prevenir errores y comportamientos inesperados. Esto incluye verificar que las estaciones, líneas y opciones ingresadas sean válidas y coherentes.

-Manejo de errores y excepciones: El programa debe estar preparado para manejar errores y excepciones de manera adecuada, proporcionando mensajes claros al usuario en caso de problemas y evitando que el programa se bloquee o produzca resultados incorrectos.

Optimización de rendimiento: Es importante optimizar el rendimiento del programa mediante el uso de algoritmos eficientes y estructuras de datos adecuadas. Esto garantiza que el programa pueda manejar grandes cantidades de datos de manera eficiente.

-Documentación y comentarios: Se debe proporcionar una documentación clara y comentarios en el código para facilitar la comprensión y el mantenimiento del programa por parte de otros desarrolladores en el futuro.