UNIVERSIDAD DE ANTIOQUIA
Facultad de Ingeniería

Informática II
Semestre 2024 – 1
Desafío 2. Abril 29 de 2024

NOMBRES:

Alejandro Naranjo Naranjo
Juan Pablo Rivero Garay.

PUNTOS DEL INFORME:

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

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.

c. Algoritmos implementados debidamente intra-documentados.

d. Problemas de desarrollo que afrontó.

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

A. ANÁLISIS DEL PROBLEMA Y CONSIDERACIONES PARA LA ALTERNATIVA DE SOLUCIÓN PROPUESTA.

1. Modelo de Datos: Se necesita un diseño que permita representar estaciones, líneas y la red de metro en su conjunto.

2. Paradigma de Programación Orientada a Objetos (POO): Utilizaremos clases para representar los elementos del sistema (estaciones, líneas, red de metro) y métodos para realizar operaciones sobre ellos.

3. Relaciones entre Clases: Cada estación puede pertenecer a una o varias líneas, pero una línea solo puede pertenecer a una red. Esto sugiere una relación de composición entre estaciones y líneas, y una relación de composición entre líneas y la red de metro.

4. Operaciones necesarias: Será necesario implementar operaciones para agregar o eliminar estaciones a líneas, agregar líneas o eliminar a la red y también una que permita calcular el tiempo .

Idea para el programa

Guardar las estaciones en punteros y las líneas en punteros dobles porque puede ofrecer ventajas en términos de flexibilidad y eficiencia en la gestión de la memoria. Aquí hay algunas consideraciones sobre esta propuesta:

1. Flexibilidad en la gestión de memoria: El uso de punteros permite una gestión más dinámica de la memoria, lo que puede ser útil si se necesitan agregar o eliminar estaciones o líneas durante la ejecución del programa.

2. Eficiencia en términos de espacio: Almacenar las líneas como punteros dobles puede reducir el espacio utilizado, ya que solo se almacenan las direcciones de memoria en lugar de los objetos completos. Esto puede ser beneficioso en sistemas con limitaciones de memoria.

Esquema del desafio

1. Clase Estación:

- Cada estación puede ser representada como un objeto de la clase `Estación`.

- La clase `Estacion` puede tener atributos como un nombre único para identificar la estación

-eliminar estación

-agregar estación

-ver si hay conexiones con esa estacion

2. Clase Línea

- clase `Linea` para representar cada línea de metro en tu red.

- Cada objeto `Linea` puede contener información sobre las estaciones que pertenecen a esa línea.

-agregar línea

-eliminar línea (hipotéticamente hablando"

- También puedes incluir información adicional como el color de la línea (no se si sea necesario hay que preguntar), el tiempo de recorrido estimado entre estaciones.

3. clase redMetro:

- clase `redmaMetro` que actúe como el controlador principal del sistema de metro.

-básicamente que muestre todo en pantalla

- líneas

- estaciones que hacen parte de una línea en especifico

- Esta clase puede contener métodos para la planificación de rutas, el cálculo de tiempos de viaje(seria lo mismo que en la clase línea).

Definir como se guardaran los datos y ver si podemos crear una red metro aleatoria como de principio o una red básica y que el usuario la personalice

Bueno, una de las condiciones que se nos exigen en el desafio es que no podemos usar el STL (Standard Template Library) para este desafio, por lo cual, una de las soluciones para, aparte mejorar lo que sería, hablando en términos de eficiencia, se nos ocurrió como crear 2 clases, que "simulen" lo que hacen los vectores y las listas de la STL.

Las clases para el vector y la lista se crean para como hacer una versión mas

personalizada y que hagan una especie de simulación de su funcionamiento, con esto, nos brindan un mayor control y flexibilidad.

Imagina que estás construyendo un juego de bloques Lego desde cero en lugar de usar uno prehecho. Al hacer esto, podemos diseñar los bloques Lego para que se ajusten exactamente a lo que necesitamos para nuestro proyecto, como agregar más características o cambiar cómo se conectan entre sí. En términos más simples, es como crear tus propias herramientas en lugar de usar las que ya están disponibles, lo que te permite adaptarlas a tus necesidades específicas.

Estas clases tiene las siguientes funciones:

Clase para el Vector (MiVector):

- Permite almacenar una secuencia de elementos del mismo tipo de manera dinámica.
-
- Proporciona funciones para agregar elementos al final de la secuencia
-
- Controla la capacidad y el tamaño del vector
-
- Puede incluir otras funciones típicas de un vector, como eliminar elementos, acceder a elementos específicos (operator[]), entre otras

Clase para la Lista (MiLista):

- Representa una estructura de datos enlazada donde cada elemento está conectado al siguiente.
-
- Permite agregar elementos al final de la lista o al principio
-
- Ofrece la capacidad de eliminar elementos
-
- Proporciona funciones para recorrer la lista y acceder a elementos
- específicos.

1. En resumen, estas clases funcionan de manera similar a los vectores y listas estándar de C++, pero están diseñadas para adaptarse mejor a las necesidades específicas de un proyecto y para evitar el uso directo de las estructuras estándar del lenguaje.

   Bueno, luego de analizarlo junto con mi compañero, llegamos a la conclusión de que esto no era una buena práctica hablando en cuanto a la abstracción de nuestro código, por lo que optamos nuevamente por solo usar 3 clases:

   RedMetro
   Linea
   Estaciones

   Uno de los mayores retos o dificultades que nos presento fue acerca de la estructura de datos, como guardas los datos y poder navegar a través de ellos sin el uso del STL, por lo que decidimos optar nuevamente por los punteros, cosa que lo hace muy eficiente y muy práctico diría yo.
   Estos, como en la practica 1, apuntas hacia arreglos.

   Gestión Dinámica de Estaciones y Líneas:

   - Al utilizar un enfoque sin el uso directo de estructuras de datos estándar como los vectores o listas de la STL, podríamos optar por gestionar dinámicamente las estaciones y líneas del sistema de metro mediante el uso de punteros y dobles punteros.

2. **Doble Puntero para Estaciones en Líneas**:

   - Podríamos tener un doble puntero que apunte a un array de punteros a estaciones para cada línea de metro. Esto nos permitiría gestionar dinámicamente la cantidad de estaciones en cada línea y realizar operaciones como agregar o eliminar estaciones sin tener un tamaño fijo predeterminado.

3. **Uso en la Clase Linea**:

- En la clase Linea, podríamos tener un atributo que sea un doble puntero a punteros de Estacion, por ejemplo:

4. **Inicialización Dinámica de Estaciones**:
    - Al agregar una nueva estación a una línea, podríamos asignar memoria dinámicamente para un nuevo puntero a Estacion y luego actualizar el doble puntero en la línea para incluir esta nueva estación.
5. **Eliminación de Estaciones**:
    - Para eliminar una estación de una línea, podríamos liberar la memoria del puntero a esa estación y luego reorganizar el array de punteros a estaciones en el doble puntero de la línea para eliminar la estación deseada.
6. **Ventajas de la Gestión Dinámica**:
    - Este enfoque nos proporciona flexibilidad en la gestión de estaciones y líneas, ya que no estamos limitados por un tamaño fijo predefinido para las estructuras de datos. Podemos adaptarnos dinámicamente a la cantidad de estaciones y líneas que el sistema de metro requiera.

Clase Estacion:

Atributos:

- **nombreEstacion**: Es un puntero a cadena de caracteres (String) que almacena el nombre de la estación. Al utilizar un puntero, podemos gestionar dinámicamente la memoria asignada para el nombre de la estación, lo que permite una flexibilidad en la longitud del nombre.
- **tiempoSiguiente**: Es un entero (int) que representa el tiempo en segundos hasta la siguiente estación. Esta información es crucial para calcular los tiempos de llegada entre estaciones dentro de una línea de metro.

- **tiempoAnterior**: Similar al atributo anterior, es un entero que indica el tiempo en segundos hasta la estación anterior. Esta información es útil para determinar los tiempos de recorrido y planificar los trayectos en el simulador de metro.
- **esTransferencia**: Un booleano (bool) que indica si la estación es de transferencia o no. Esta característica es importante para determinar las posibilidades de conexión entre líneas y facilitar la navegación de los usuarios en el sistema de metro simulado.

Métodos:

- **Estacion(nombre, siguiente, anterior, transferencia)**: Constructor de la clase Estacion que inicializa los atributos al crear una nueva estación. Este constructor es esencial para la creación y configuración inicial de cada estación en el simulador.
- **esTransferencia()**: Método que devuelve true si la estación es de transferencia y false en caso contrario. Esta función es útil para realizar verificaciones y tomar decisiones basadas en el tipo de estación en diferentes partes del simulador.

Getters y Setters:

- **getNombreEstacion()**: Retorna el nombre de la estación como un puntero a cadena de caracteres (String). Permite acceder al nombre de la estación desde fuera de la clase para su visualización o manipulación.
- **setNombreEstacion(nombre)**: Establece el nombre de la estación utilizando un puntero a cadena de caracteres como parámetro. Este método es útil para actualizar el nombre de una estación si es necesario.
- **getTiempoSiguiente()**: Retorna el tiempo hasta la siguiente estación como un entero (int). Permite obtener información sobre el tiempo de viaje entre estaciones para calcular los tiempos de llegada.
- **setTiempoSiguiente(tiempo)**: Establece el tiempo hasta la siguiente estación utilizando un entero como parámetro. Es útil para actualizar los tiempos de viaje entre estaciones según sea necesario.
- **getTiempoAnterior()**: Retorna el tiempo hasta la estación anterior como un entero (int). Proporciona información sobre los tiempos de recorrido en el sentido opuesto al habitual.
- **setTiempoAnterior(tiempo)**: Establece el tiempo hasta la estación anterior utilizando un entero como parámetro. Útil para ajustar los tiempos de recorrido en direcciones específicas.

- **getEsTransferencia()**: Retorna un booleano que indica si la estación es de transferencia o no. Permite verificar la naturaleza de la estación para decisiones de enrutamiento y navegación.
- **setEsTransferencia(transferencia)**: Establece si la estación es de transferencia o no utilizando un booleano como parámetro. Útil para marcar o desmarcar estaciones según sea necesario en el sistema de metro simulado.

Clase Linea:

Atributos:

- **nombreLinea**: Es un puntero a cadena de caracteres (String) que almacena el nombre de la línea de metro. Utilizar un puntero permite gestionar dinámicamente la memoria asignada para el nombre de la línea, lo que facilita la flexibilidad en la longitud del nombre.
- **estaciones**: Es un arreglo dinámico de punteros a objetos de la clase Estacion. Este arreglo permite almacenar las estaciones que pertenecen a la línea de metro, asegurando un manejo eficiente de la memoria y la capacidad de agregar o eliminar estaciones según sea necesario.
- **cantidadEstaciones**: Un entero que representa la cantidad actual de estaciones en la línea de metro. Este atributo es útil para realizar operaciones que involucren el número de estaciones en la línea, como obtener la cantidad o determinar si la línea está vacía.

Métodos:

- **Linea(nombre)**: Constructor de la clase Linea que inicializa el nombre de la línea y el arreglo de estaciones. Este constructor es esencial para crear una nueva línea de metro y configurar sus atributos iniciales.
- **agregarEstacion(estacion)**: Método que agrega una estación a la línea de metro. Utiliza un puntero a Estacion como parámetro para evitar copias y garantizar la eficiencia en el manejo de memoria. Este método es crucial para la operación del simulador al añadir nuevas estaciones a una línea existente.
- **eliminarEstacion(nombre)**: Método que elimina una estación de la línea de metro según su nombre. Actualiza el arreglo de estaciones de manera eficiente, eliminando la estación deseada de la línea. Este método es importante para mantener la integridad y coherencia de las líneas de metro.

- **obtenerCantidadEstaciones()**: Método que devuelve la cantidad actual de estaciones en la línea de metro. Permite obtener información sobre el tamaño de la línea y su evolución a lo largo del tiempo.

Getters y Setters:

- **getNombreLinea()**: Retorna el nombre de la línea como un puntero a cadena de caracteres (String). Permite acceder al nombre de la línea desde fuera de la clase para su visualización o manipulación.
- **setNombreLinea(nombre)**: Establece el nombre de la línea utilizando un puntero a cadena de caracteres como parámetro. Este método es útil para actualizar el nombre de una línea de metro si es necesario.
- **getEstaciones()**: Retorna el arreglo de estaciones como un arreglo dinámico de punteros a objetos de la clase Estacion. Permite acceder a las estaciones de la línea para su visualización o manipulación.
- **getCantidadEstaciones()**: Retorna la cantidad actual de estaciones en la línea de metro como un entero (int). Proporciona información sobre el tamaño de la línea y su evolución.

Clase RedMetro:

Atributos:

- **lineas**: Es un arreglo dinámico de punteros a objetos de la clase Linea que representa las líneas de metro en la red. Utilizar un arreglo dinámico permite gestionar eficientemente la memoria asignada para las líneas de metro y la capacidad de agregar o eliminar líneas según sea necesario.
- **cantidadLineas**: Un entero que representa la cantidad actual de líneas en la red de metro. Este atributo es útil para realizar operaciones que involucren el número de líneas en la red, como obtener la cantidad o determinar si la red está vacía.

Métodos:

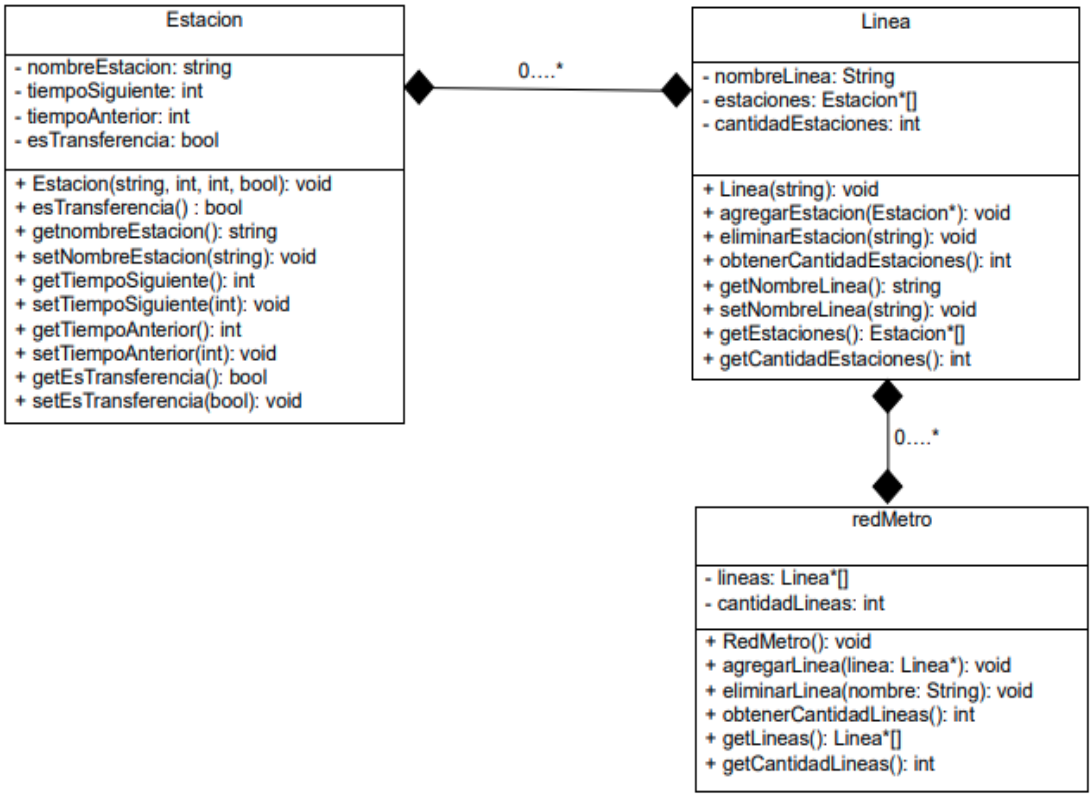
- **RedMetro()**: Constructor de la clase RedMetro que inicializa el arreglo de líneas. Este constructor es esencial para crear la red de metro y configurar sus atributos iniciales.

- **agregarLinea(linea)**: Método que agrega una línea a la red de metro. Utiliza un puntero a Linea como parámetro para evitar copias y gestionar eficientemente la memoria. Este método es crucial para la operación del simulador al añadir nuevas líneas a la red existente.
- **eliminarLinea(nombre)**: Método que elimina una línea de la red de metro según su nombre. Actualiza el arreglo de líneas de manera eficiente, eliminando la línea deseada de la red. Este método es importante para mantener la integridad y coherencia de la red de metro.
- **obtenerCantidadLineas()**: Método que devuelve la cantidad actual de líneas en la red de metro. Permite obtener información sobre el tamaño de la red y su evolución a lo largo del tiempo.

Getters y Setters:

- **getLineas()**: Retorna el arreglo de líneas como un arreglo dinámico de punteros a objetos de la clase Linea. Permite acceder a las líneas de la red para su visualización o manipulación.
- **getCantidadLineas()**: Retorna la cantidad actual de líneas en la red de metro como un entero (int). Proporciona información sobre el tamaño de la red y su evolución.

B. DIAGRAMA DE CLASES DE LA SOLUCIÓN PLANTEADA.
ADICIONALMENTE, DESCRIBA EN ALTO NIVEL LA LÓGICA DE LAS TAREAS QUE USTED DEFINIÓ PARAAQUELLOS SUBPROGRAMAS CUYA SOLUCIÓN NO SEA TRIVIAL.



Agregar una estación a una línea:

En la clase Linea, se implementa un método agregarEstacion(Estacion* estacion) que añade una estación al final del arreglo dinámico de estaciones de la línea. Se utiliza un puntero a la estación para evitar copias y garantizar la eficiencia en el manejo de memoria.

Eliminar una estación de una línea:

En la clase Linea, se implementa un método eliminarEstacion(string nombre) que busca y elimina una estación por su nombre del arreglo dinámico de estaciones de la línea. Esto implica liberar la memoria ocupada por la estación eliminada y ajustar el tamaño del arreglo de estaciones.

Saber cuántas líneas tiene una red Metro:

En la clase RedMetro, se implementa un método obtenerCantidadLineas() que devuelve el número actual de líneas en la red de metro. Este valor se actualiza dinámicamente a medida que se agregan o eliminan líneas de la red.

Saber cuántas estaciones tiene una línea dada:

En la clase Linea, se implementa un método obtenerCantidadEstaciones() que retorna la cantidad actual de estaciones en la línea. Este valor se actualiza automáticamente al agregar o eliminar estaciones.

Saber si una estación dada pertenece a una línea específica:

Podría implementarse un método en la clase Linea para buscar una estación por su nombre y determinar si pertenece a esa línea específica.

Agregar una línea a la red Metro:

En la clase RedMetro, se implementa un método agregarLinea(Linea* linea) que añade una nueva línea al arreglo dinámico de líneas de la red de metro. Se utiliza un puntero a la línea para evitar copias y gestionar eficientemente la memoria.

Eliminar una línea de la red Metro:

En la clase RedMetro, se implementa un método eliminarLinea(string nombre) que busca y elimina una línea por su nombre del arreglo dinámico de líneas de la red. Esto implica liberar la memoria ocupada por la línea eliminada y ajustar el tamaño del arreglo de líneas.

Saber cuántas líneas tiene la red:

En la clase RedMetro, se implementa un método obtenerCantidadLineas() que devuelve la cantidad actual de líneas en la red de metro. Este valor se actualiza dinámicamente al agregar o eliminar líneas.

Saber cuántas estaciones tiene una red Metro:

En la clase RedMetro, se implementaria un método que recorra todas las líneas y sume la cantidad de estaciones de cada una, teniendo en cuenta que una estación de transferencia podría contar como una sola estación aunque pertenezca a múltiples líneas.

D. PROBLEMAS DE DESARROLLO QUE AFRONTÓ:

Bueno, yo creo que el problema que mas nos costo fue con el manejo de los datos, ya que como lo dije antes, en el enunciado del desafio no nos permitían usar el STL, y pues en parte entiendo porque si no se perdería la gracia de llamarlo desafio, porque en realidad seria todo como muy trivial.
Y creo que también nos costo un poquito el tema de la abstracción, porque si bien puede parecer un tema sencillo, a la hora de enfrentarse a un problema, no lo es tanto.

E. EVOLUCIÓN DE LA SOLUCIÓN Y CONSIDERACIONES PARA TENER EN CUENTA EN LA IMPLEMENTACIÓN.

Yo creo que tuvimos una buena evolución en toda esta semana que llevamos analizando el problema, si bien la idea inicial era declarar 2 clases mas que hicieran como una especie de simulación de lo que seria la STL, luego nos lo replanteamos y nos parecía que había una mejor forma de hacer esto, ya que si nos dijeron que no usaramos la STL, es porque creo que los profesores buscaban queviéramos y buscáramos un enfoque diferente.

Ahora bien, este no es el proyecto como tal y definitivo, seguramente, en esta semana restante que nos queda, podamos ir viendo que hay mejoras por corregir, por lo que esta no es una versión 100% concluida y definida del desafio #2.

Muchas gracias por la atención prestada!.