



Universidad de Antioquia

Facultad de Ingeniería

Programas educativos:

**Ingeniería de Telecomunicaciones
Ingeniería Electrónica**

Docentes:

ANÍBAL GUERRA

AUGUSTO SALAZAR

Parcial 2: Informática II

Desafío 1

Informe para Desafío 2

Estudiantes:

ANTONIO CARLOS MERLANO RICARDO (1010152199)

JUAN ESTEBAN LOPEZ CASTRILLON (1127941113)

Análisis del simulador de red metro.

Para abordar este desafío del simulador de red de metro, buscamos desarrollar un sistema que nos permita modelar la estructura y el funcionamiento de un sistema de metro. La idea será crear un programa que simule cómo operan las estaciones, líneas y trenes dentro de una red de metro.

Como primeramente vamos a diseñar un modelo de datos, utilizando la programación orientada a objetos. Esto significa que vamos a crear clases para representar cada componente del sistema: estaciones, líneas y la red de metro en general. Cada una de estas clases tendrá atributos y métodos para gestionar las operaciones que necesitamos realizar, como agregar o eliminar estaciones y líneas, verificar la pertenencia de una estación a una línea, entre otras.

La estructura de nuestro sistema reflejará la relación jerárquica entre estaciones, líneas y la red de metro. Por ejemplo, una estación puede pertenecer a varias líneas, pero una línea solo puede pertenecer a una red.

Además, debemos implementar una función que simule el movimiento de los trenes entre estaciones. Esta función calculará el tiempo que tarda un tren específico en llegar de una estación a otra dentro de la misma línea. Utilizaremos la hora actual como punto de partida del tren y tendremos en cuenta los tiempos de llegada entre estaciones para hacer este cálculo.

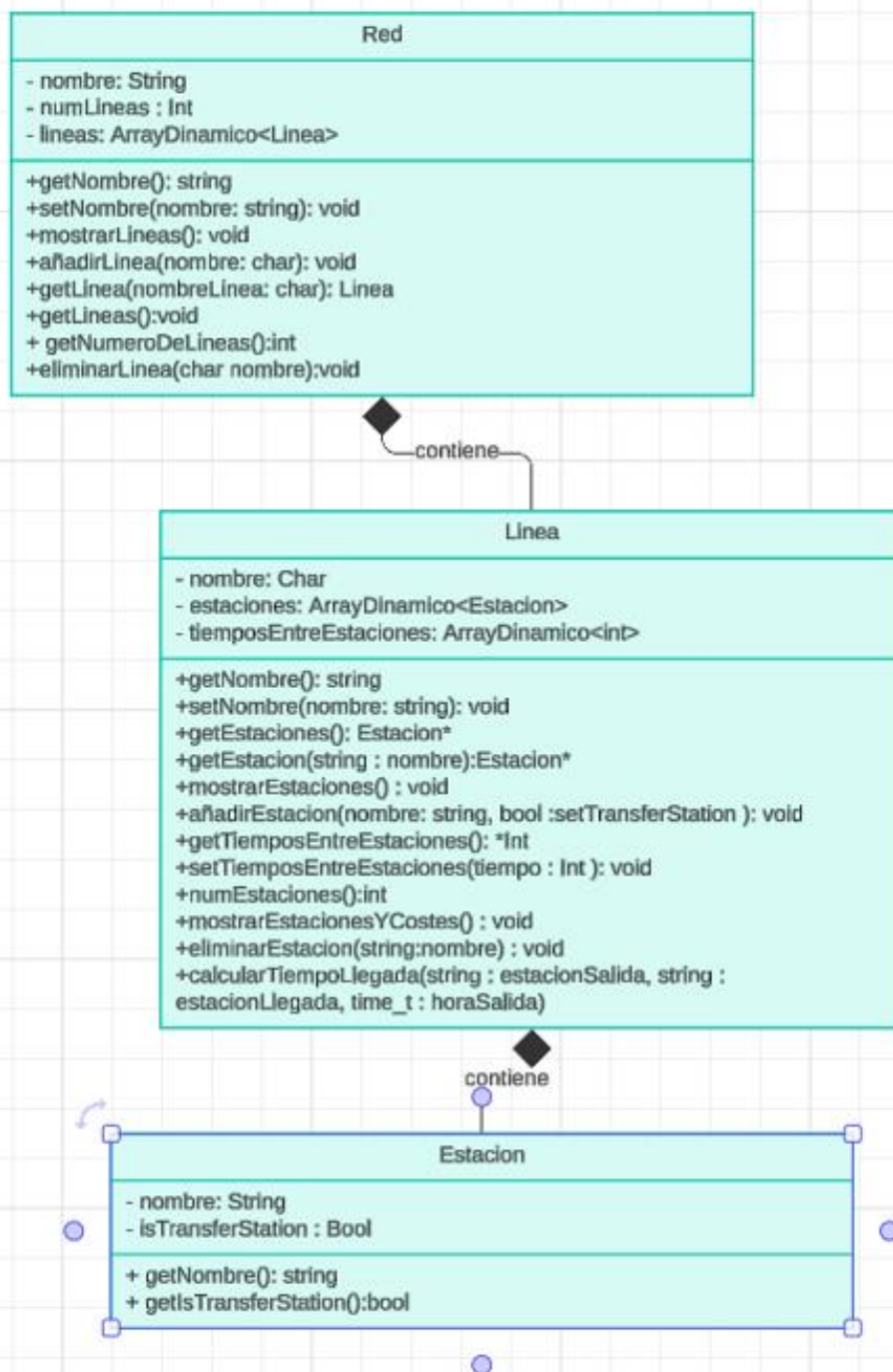
Para interactuar con nuestro simulador, vamos a crear una interfaz de usuario simple. Esto incluirá el menú que permitirá al usuario realizar diferentes acciones, como agregar estaciones y líneas, simular el tiempo de llegada de un tren, entre otras cosas. Nos aseguramos de que la interfaz sea sencilla y que guíe al usuario a través de las diferentes funcionalidades del simulador.

Finalmente, realizaremos pruebas exhaustivas nos aseguramos de que nuestro simulador funcione correctamente en diferentes situaciones. Vamos a verificar que todas las operaciones se realicen correctamente y que el simulador cumpla con los requisitos establecidos por el cliente.

Solución propuesta

Se requirió encontrar una solución alternativa que permitiera simular el comportamiento de los vectores dinámicos en tiempo de ejecución, para poder manejar de manera eficiente los datos relacionados con las líneas y estaciones del metro. Lo cual llevo a desarrollar una función que genera arreglos dinámicos y simular el comportamiento de los vectores. Esta función recibe un nuevo dato para ser insertado, el puntero al arreglo dinámico antiguo y el tamaño de este, y realiza las operaciones necesarias para insertar el nuevo dato y redimensionar el arreglo. Pero al final optamos por el medio de uso de plantillas de clases para crear una clase para esto, esta clase podíamos hacer lo mismo que en la función y además poder tener un atributo que nos almacenara el tamaño del arreglo, ya que con esto tendríamos nuestro propio arreglo dinámico o vector hecho para finalmente gracias a la sobrecarga de operadores simulamos como si accediéramos a sus elementos como un arreglo normal.

Diagrama de clases



Tenemos la clase red que tiene unos atributos y unos métodos además tiene los métodos normales que son los getters y los setters algunos métodos superimportantes para la función para la clase red es el método que nos permite obtener las líneas que pertenece a la red. Luego tenemos una clase línea que también tiene sus atributos y sus métodos con sus getters y setters algunos métodos importantes son el método de añadir estación el método de get estación que nos permite obtener una estación A partir de ese arreglo estaciones que tenemos en esa línea eh y otros métodos que nos sirven para calcular el tiempo de llegada mostrar estaciones y sus costos etcétera.

y en la clase estación no nos enfocamos tanto en sus métodos y atributos ya que una estación es una clase fácil de abstraer porque una estación no contiene objetos por así decirlo como la clase red y la clase línea ya que la red tenía líneas y las líneas contienen estaciones yo creo que lo más difícil fue lograr hacer Que cada una de esas clases objetos tuviera relación con la otra

Algoritmos implementados

En nuestro desarrollo del desafío se implementamos muchos algoritmos y funciones para nuestro desarrollo algunas de las más importante para simplificar y cumplir con nuestro objetivo del desafío fueron las declaraciones dentro de los archivos red.h y línea.h, a continuación, como definimos nuestras clases y funciones para estos dos archivos

1. Para la clase Red la usamos para gestionar una red de líneas, el desarrollo de estas funciones dentro de esta clase se encuentra en el archivo red.cpp

```

#ifndef RED_H
#define RED_H
#include<string>
#include "ArrayDinamico.h"
#include "linea.h"
using namespace std ;

class Red
{
private:
    string nombre;
    int numLineas = 0;
    ArrayDinamico<Linea> lineas;

public:
    Red(string nombre);
    string getNombre();
    void mostrarLineas();
    void setNombre(string nombre);
    void anadirLinea(char nombre);
    Linea* getLinea(char nombre);
    Linea* getLineas();
    int getNumeroDeLineas();
    void eliminarLinea(char nombre);
};

```

- Red::Red(string nombre): Este es el constructor de la clase Red, que toma un parámetro nombre y lo asigna al atributo nombre de la red.
 - string Red::getNombre(): Este método devuelve el nombre de la red.
 - void Red::setNombre(string nombre): Este método establece el nombre de la red.
 - void Red::mostrarLineas(): Este método muestra todas las líneas asociadas a la red, mostrando el nombre de cada una.
 - void Red::anadirLinea(char nombre): Este método agrega una nueva línea a la red. Toma un parámetro nombre que representa el nombre de la línea.
 - Linea* Red::getLinea(char nombre): Este método busca una línea en la red por su nombre y devuelve un puntero a ella si se encuentra. Si no se encuentra, devuelve null.
 - int Red::getNumeroDeLineas(): Este método devuelve el número total de líneas en la red.
2. El desarrollo esta implementado en un archivo línea.cpp, este archivo línea lo utilizamos para gestionar y manipular las estaciones asociadas a las líneas así como los tiempos de recorridos entre estaciones.

```

#ifndef LINEA_H
#define LINEA_H
#include<string>
#include"ArrayDinamico.h"
#include"estacion.h"
using namespace std;

class Linea
{
private:
    char nombre;
    ArrayDinamico<Estacion> estaciones;
    ArrayDinamico<int> tiempoEntreEstaciones;
public:
    Linea();
    Linea(char nombre);
    char getNombre();
    //metodo para obtener las estaciones de las lineas (retorna la memoria del primer elemento del arreglo)
    Estacion* getEstaciones();
    //metodo para obtener una estacion;

    Estacion* getEstacion(string nombre);

    //anadirEstacion
    void anadirEstacion(string nombre, bool setTransferStation);

    //metodo para obtener los tiempos entre estaciones
    int *getTiempoEntreEstaciones();

    //metodo que me retorna el numero de estaciones de la linea

    int numEstaciones();

    //metodo para imprimir las estaciones
    void mostrarEstaciones();

    void mostrarEstacionesYCostes();

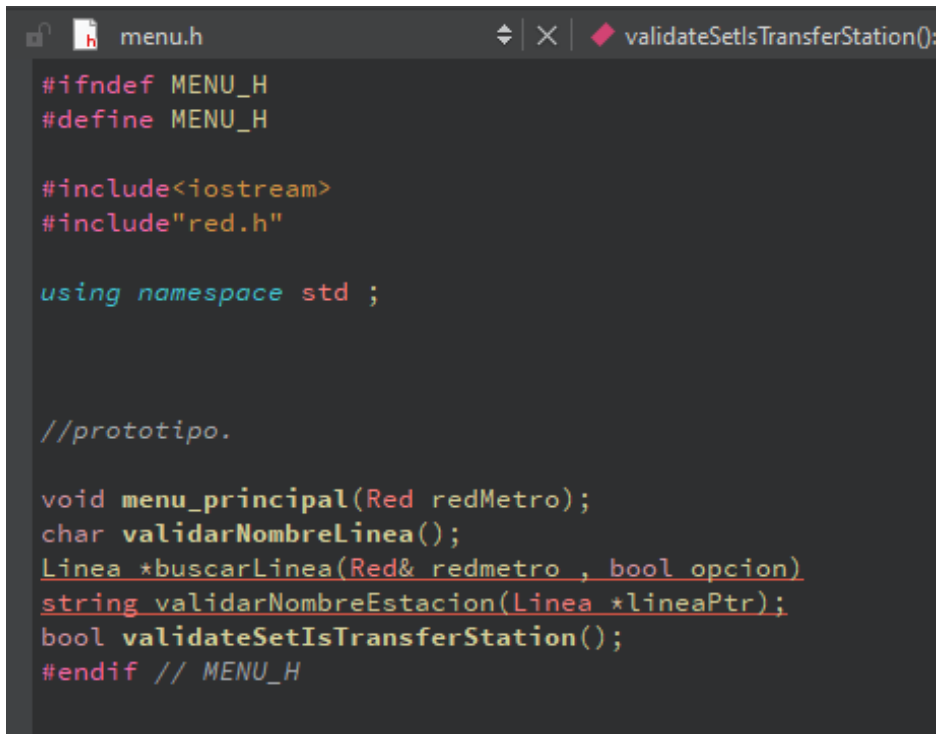
    void eliminarEstacion(string nombre);
};

#endif // LINEA_H

```

- Linea::Linea(): Este es el constructor por defecto de la clase Linea. Inicializamos el atributo nombre de la clase con un valor predeterminado.
- Linea::Linea(char nombre): Este es otro constructor de la clase Linea, toma un parámetro nombre y lo asigna al atributo nombre de la clase.
- char Linea::getNombre(): Este método devuelve el nombre de la línea.
- Estacion* Linea::getEstaciones(): Este método devuelve un puntero a la primera estación en la lista de estaciones asociadas a la línea.
- void Linea::eliminarEstacion(string nombre): Este método elimina una estación de la lista de estaciones asociadas a la línea, dado su nombre. También ajusta los tiempos de recorrido entre estaciones si es necesario.
- int* Linea::getTiempoEntreEstaciones(): Este método devuelve un puntero al primer elemento en la lista de tiempos de recorrido entre estaciones.
- void Linea::anadirEstacion(string nombre, bool setTransferStation): Este método añade una nueva estación a la lista de estaciones asociadas a la línea. También solicita al usuario los tiempos de recorrido entre la nueva estación y las estaciones adyacentes, si corresponde.
- void Linea::mostrarEstaciones(): Este método muestra una lista de estaciones asociadas a la línea.
- void Linea::mostrarEstacionesYCostes(): Este método muestra una lista de estaciones asociadas a la línea junto con los tiempos de recorrido entre ellas.
- int Linea::numEstaciones(): Este método devuelve el número de estaciones asociadas a la línea.

3. Creamos un menú interactivo para gestionar una red de transporte que incluya la adición eliminación de líneas y estaciones en la red. Donde aquí las declaramos y en menú.cpp desarrollamos



```
#ifndef MENU_H
#define MENU_H

#include<iostream>
#include"red.h"

using namespace std ;

//prototipo.

void menu_principal(Red redMetro);
char validarNombreLinea();
Linea *buscarLinea(Red& redmetro , bool opcion)
string validarNombreEstacion(Linea *lineaPtr);
bool validateSetIsTransferStation();
#endif // MENU_H
```

- menu_principal(Red redMetro): Esta función representa el menú principal del programa. Permite al usuario realizar varias operaciones, como agregar una línea, agregar una estación, eliminar una estación, etc.
 - char validarNombreLinea(): Esta función valida y devuelve el nombre de una línea ingresado por el usuario.
 - Linea *buscarLinea(Red& redmetro, bool opcion): Esta función busca una línea en la red según el nombre ingresado por el usuario. Dependiendo del valor de opcion, puede ser para agregar o eliminar una estación.
 - string validarNombreEstacion(Linea *lineaPtr): Esta función valida y devuelve el nombre de una estación ingresado por el usuario, asegurándose de que el nombre no esté duplicado en la línea.
 - bool validateSetIsTransferStation(): Esta función valida si una estación es una estación de transferencia, es decir, una estación donde los pasajeros pueden cambiar de línea.
4. La clase ArrayDinamico.h implementa un arreglo dinámico que puede crecer y decrecer en tamaño según sea necesario. Se creo porque:
 1. plantillas (templates): La clase ArrayDinamico es una plantilla de clase que toma un parámetro de tipo (T). Esto permite crear instancias de la clase ArrayDinamico con diferentes tipos de elementos según sea necesario.
 2. Métodos de la clase:

- `anadir(T elemento)`: Añade un elemento al arreglo dinámico. Incrementa el tamaño del arreglo y copia los elementos existentes a un nuevo arreglo más grande.
- `insertarEn(T elemento, int indice)`: Inserta un elemento en una posición específica del arreglo. Similar a `anadir`, pero permite especificar el índice de inserción.
- `eliminarEn(int indice)`: Elimina un elemento del arreglo en la posición especificada y ajusta el tamaño del arreglo.

3. Gestión de memoria dinámica: El constructor por defecto inicializa el puntero a `nullptr` y el tamaño a 0. Los métodos de la clase gestionan la memoria dinámicamente utilizando `new` para asignar memoria para el arreglo y `delete[]` para liberarla cuando ya no se necesite.

4. Operador de indexación sobrecargado (`operator[]`): Permite acceder a los elementos del arreglo utilizando la notación de corchetes (`[]`). Este operador devuelve una referencia al elemento en la posición especificada, lo que permite tanto la lectura como la escritura de elementos en el arreglo.

5. Métodos adicionales: La clase también proporciona métodos para obtener el tamaño del arreglo, mostrar los elementos del arreglo y obtener la dirección de memoria del primer elemento del arreglo.

```
public:
    ArrayDinamico() {
        this->arrayPtr = nullptr;
        this->size = 0;
    }

    /*~ArrayDinamico() {
        cout<<"eliminando...";
        delete[] arrayPtr;
    }*/

    //metodos para añadir un elemento a mi arreglo dinamico
    void anadir(T elemento) {
        //creamos un nuevo arreglo dinamico del tamaño del arreglo que teniamos mas 1 para poder insetar el nuevo elemento
        T* newArray = new T[size + 1];
        //copiamos el arreglo viejo al nuevo
        for (int i = 0; i < size; i++) {
            newArray[i] = arrayPtr[i];
        }

        //ahora agregamos el ultimo elemento al nuevo arreglo
        newArray[size] = elemento;
        delete[] arrayPtr;
        //cout<<"acabo de eliminar el array viejo " ;
        // y ahora hacemos que el puntero arrayPtr apunte al nuevo arreglo
        arrayPtr = newArray;
        //y por ultimo modificamos el atributo de size
        size++;
    }
```

Problemas en el desarrollo:

Primeramente, nuestro problema inicial fue como íbamos a estructurar nuestro programa y que estructuras usaríamos al no poder usar las librerías STL y los contenedores ya que no podríamos almacenar las líneas de la red y las estaciones en base a esto decidimos crear nuestra propia estructura de datos similar al vector gracias a el template que nos ofrece c++.

Al implementar el método añadir estación entre estaciones fue otro inconveniente porque teníamos que crear un arreglo nuevo con la longitud del arreglo anterior y sumarle uno y luego copiarlo a ese punto y ya luego en el punto que había que ingresar esa estación meter la estación para copiar el arreglo antiguo.

Eliminar la estación fue un problema, pero no menor, le dimos solución copiando el arreglo tal cual y no le incluimos el elemento del arreglo anterior.

Evolución del programa

Para el desarrollo en qt dividimos el proyecto en varios archivos el main.cpp, red.cpp, estacion.cpp, línea.cpp, menú.cpp. Aclarando que en los .h de estos mismos archivos con los mismos nombre declaramos nuestras clases y funciones y en los .cpp el desarrollo de estas.

En lo que se desarrolló el programa se llevó paso a paso por la guía, a continuación, la evolución del programa por serie de pasos: lo primero fueron las implementaciones de los métodos a medida que íbamos evolucionando nos tocaba hacer o eliminar implementaciones que ya íbamos aprobando o descartando.

1. **Métodos de la Red:** Implementa una clase Red que representa una red de transporte, permitiendo la administración y gestión de líneas dentro de esa red. Con métodos para agregar líneas, obtener información sobre las líneas existentes, como su nombre y número, y buscar líneas específicas por su nombre, la clase facilita la manipulación y visualización de la red y sus componentes.
2. **Método de líneas:** Esta implementación proporciona funcionalidades completas para gestionar una línea de transporte, desde la adición y eliminación de estaciones hasta la visualización de información detallada sobre la línea y sus estaciones.
3. **Métodos de estaciones:** Proporciona una clase básica para representar estaciones en un sistema de red, permitiendo acceder a su nombre y a su estado de estación de transferencia.

De manera progresiva complementamos nuestro desarrollo del desafío como por ejemplo la creación del menú que nos permitiera la interacción con nuestro sistema de red metro. La Evolución implica muchas dificultades, análisis y momentos decisivos que llevaron a cabo con tomas de decisiones para el desarrollo de este simulador de red metro.