Informe Desafío 2 – Juan Pablo de Jesus Avendaño Bustamante

Análisis del problema

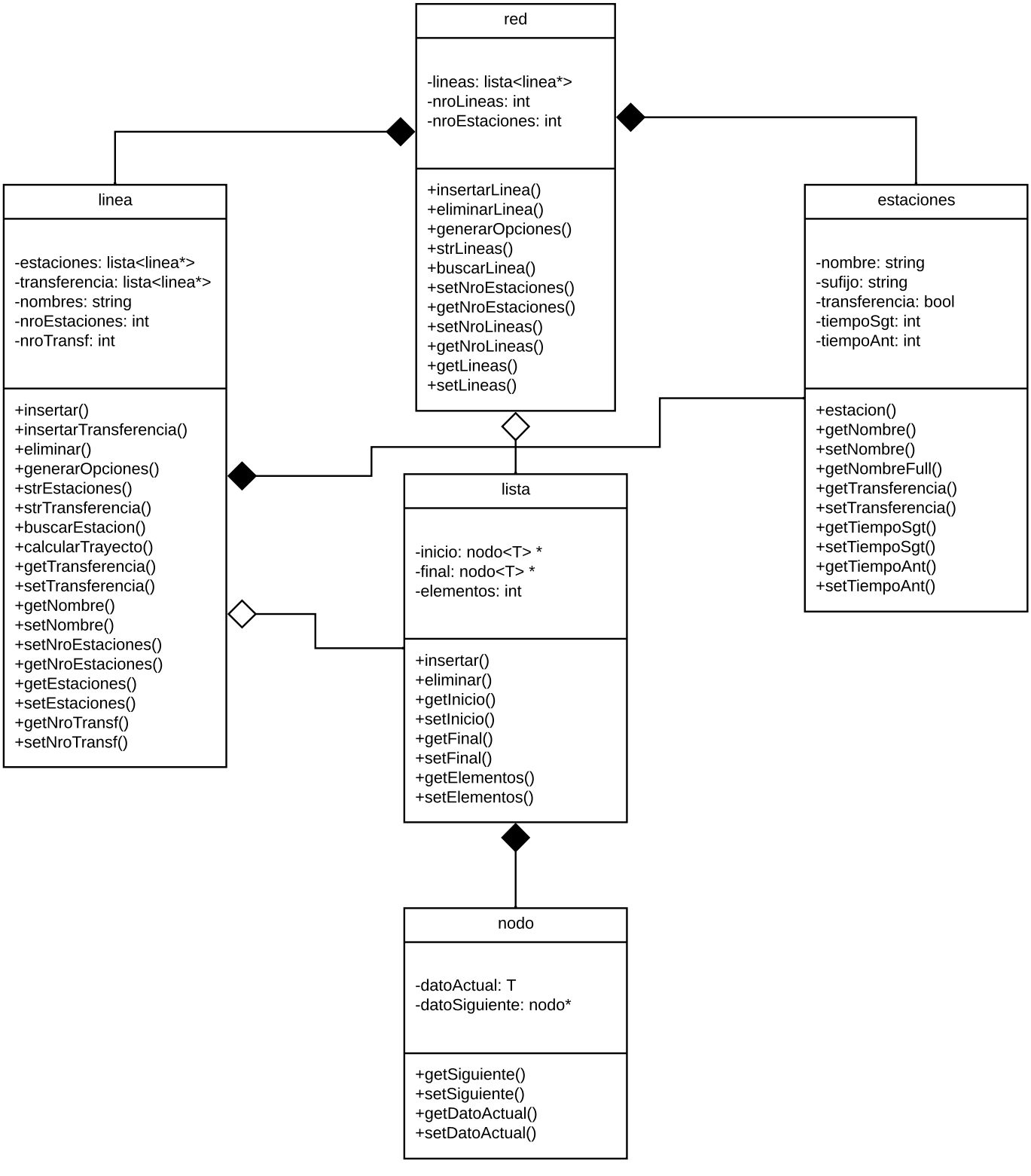
Durante el análisis del problema se hizo necesaria la implementación de la programación orientada a objetos, para construir la red de metro se plantearon tres clases principales que permitirían acceder de manera ordenada a cada línea o estación, siendo así que se introdujeron las clases red, línea y estación, cada una de ellas con sus métodos y propiedades característicos. Debido a la naturaleza del problema se hizo imperativo implementar una estructura de datos que permitiera realizar búsqueda, inserción y eliminación de manera eficiente, a raíz de esta necesidad se implementó una plantilla que iba a contener una lista enlazada con sus respectivos nodos, en cuanto a eficiencia se refiere, se intercambió eficiencia a la hora de buscar (búsqueda en una lista simplemente enlazada tiene complejidad de O(n)) por eficiencia a la hora de insertar y eliminar (solamente basta con modificar un atributo de un nodo para insertar o eliminar). Teniendo todos los elementos necesarios se empezó a definir cada uno de los métodos y los atributos de las clases.

En la clase red se introdujeron propiedades que permitieran saber el número de líneas de la red y las estaciones que poseía la red, por otro lado, en la clase línea se iba a guardar el nombre de la línea, el número de estaciones de la línea, tanto estaciones de transferencia como estaciones totales, además de eso se definieron métodos que permitían buscar una estación en una línea y calcular el tiempo que se demora de una estación a otra, por último en la clase estación se incluyeron los atributos que permitían conocer el nombre de la estación, su sufijo (nombre de la línea a la que pertenecía en el caso de que fuera de transferencia), si es de transferencia o no y el tiempo que tardaba en llegar a la estación anterior y a la siguiente. Cabe recalcar que cada clase posee sus setters y getters correspondientes.

Con este diseño de clases se tiene facil acceso a la información de cada línea o cada estación de la red, a su vez se cumplen los requisitos que hablaban sobre la existencia de una estación en

una línea y el cálculo de tiempo entre estaciones. Note que estas clases se realizaron de manera genérica, es decir, no tuvieron en cuenta las características de la simulación, estas consideraciones se hicieron en el main y no en las clases, esto apuntando a que la idea de una clase es representar de manera general un aspecto de la realidad.

DIAGRAMA DE CLASES
DESAFIO II



Como se puede ver en el diagrama de clases las relaciones entre las clases son en su mayoría de composición, de manera conceptual esto se puede ver como la existencia de un todo, del cual dependen todas las otras clases, por ejemplo hay varias clases que hacen parte de la clase red, esto quiere decir que si se elimina red, ninguna de ellas podría existir de manera separada, a su vez la lista tiene relaciones de agregación con la clase red y la clase línea, esto tiene sentido ya que la lista es un componente de las clases mencionadas que componen un todo, pero podría existir sin la presencia de aquellas clases, a su vez el nodo no podría existir sin la clase lista, dado que si se destruye una lista todos sus nodos se van con él.

Logica de subprogramas no triviales

A continuación, se detallará la logica de los subprogramas no triviales:

- **Subprograma que permite buscar una estación o línea**
  Para cada clase, línea y estación, se definió un metodo que permitía buscar si un nombre dado pertenecía a la lista que contenía a una clase u otra, lo que se hizo fue implementar una función que removiera espacios y transformara los caracteres alfabéticos de un string a minúsculas, seguido a eso se recorrió la lista perteneciente a cada parte, sea línea o red, así se iba comparando el nombre con las modificaciones realizadas que se ingresó como parámetro con los nombres guardados en cada instancia de línea o estación, cabe recalcar que en el apartado de buscar estaciones se le permite al usuario ingresar el nombre de la estación con o sin sufijo, siendo el sufijo el nombre de la línea a la cual pertenece (esto si la estación es de transferencia).

- **Subprograma que calcula el trayecto entre dos estaciones**
  Para dar solución a la necesidad de calcular el tiempo entre estaciones se planteó un subprograma alojado en la clase línea, que recibiera los índices de las estaciones de origen y destino, dentro de tal subprograma se realizaban operaciones para que el índice de origen siempre fuera menor o igual que el índice de destino, como siempre se recorría la lista de líneas, partiendo desde el origen hasta llegar al destino, se iba sumando el tiempo entre estaciones contiguas en un contador, y finalmente se

entregaba el valor de el mismo cuando se recorría la lista hasta la estación de destino.

- **Menús**
  Se realizaron subprogramas que iban a servir para la experiencia de usuario, la base de todos los menús que hay en la implementación es un ciclo en donde se le pregunta al usuario sea una opción o una entrada, se iban a aplicar criterios para decidir si lo que ingreso el usuario (existen varios menús, menú para las opciones, menú para las entradas de números, menú para ingresar nombres de líneas, y menú para ingresar nombres de estaciones). Cuando el usuario finalmente ingresa correctamente la opción el menú devuelve la entrada que el usuario ingresó en la mayoría de los casos, hay un caso de un subprograma menú que devuelve un booleano según si el usuario ingreso un nombre de una estación que está en una línea o no.

- **Agregación y eliminación de estaciones/líneas**
  El principio de agregación y eliminación de líneas resulta sencillo en la medida de que se verifica que las condiciones para la eliminación se cumplan, para realizar esto se dispone de condicionales para verificar tales condiciones mediante la llamada a métodos de cada instancia, por ejemplo, si se necesita que para eliminar una estación se verifique si es de transferencia o no, simplemente se llama al metodo que me entrega tal atributo de la estación. Hay un caso particular para la agregación de estaciones y líneas y es que si no existen estaciones o líneas no se pedirá la posicion donde se quiere ingresar o la línea con la que se quiere enlazar respectivamente.
  Para la agregación y eliminación de las partes mencionadas anteriormente se hace uso de los algoritmos de las listas enlazadas que se describirán en la siguiente sección.

Algoritmos implementados

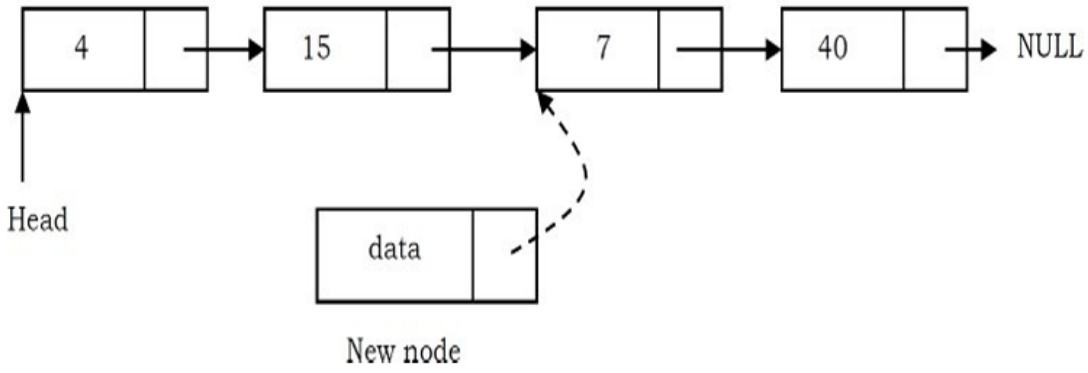
La mayoría de los algoritmos implementados tienen que ver con la estructura de datos que se encargaba de manejar los datos del programa.

- **Inserción**
  El algoritmo de inserción consta de dos partes, inserción al final o al inicio e inserción en una posición arbitraria, la inserción al final o al inicio se resuelve de manera trivial, dado que la lista posee atributos para tales posiciones específicas, esta inserción se resuelve modificando punteros según corresponda, para insertar un nodo al inicio bastaba con enlazar el nodo deseado con el nodo del inicio y seguido a eso establecer el nodo deseado como el nodo de inicio, lo que nos da una complejidad temporal de **O(1),** de forma análoga se resolvió el caso para la inserción al final.

  La solución para la inserción en una posición arbitraria desafía el criterio de eficiencia ya que tiene una complejidad **O(n)** dado que en el peor de los casos se debe recorrer toda la lista para encontrar la posición deseada, en esta sección no se pondrá en criterio la elección de esta estructura de datos, lo podrá ver al final del presente informe.

  Para poder insertar un nodo en una posición arbitraria se tiene que recorrer la estructura de datos hasta llegar una posicion antes de la deseada, en esta posicion actual se modifica el puntero al nodo siguiente del nodo a insertar, haciendo el nodo siguiente como el nodo en la posicion deseada, así entonces se modifica el puntero siguiente de la posicion actual, apuntando hacia el nodo deseado.

  Véase la siguiente imagen

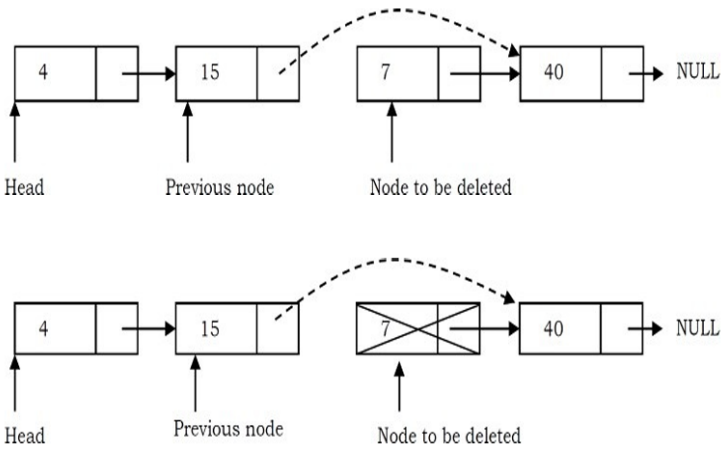


- **Eliminación**
  El algoritmo de eliminación es análogo al de inserción de datos, consiste en dos tipos de eliminación, eliminación al final o al inicio y eliminación en una posición arbitraria.

  La eliminación al inicio y al final se sigue de la inserción del mismo tipo, para cada caso nos referimos al atributo que guarda el dato en la posición de inicio o final, modificamos los punteros que sean necesarios, en el caso del inicio simplemente se asigna como nuevo inicio el nodo siguiente al inicial, y se prescinde del eliminado teniendo una complejidad de **O(1).**

  Para los casos de eliminación del objeto final y la eliminación en una posición arbitraria se desafía nuevamente el criterio de eficiencia dado que hay que recorrer la lista hasta una posicion anterior de la deseada y poner como nodo siguiente al de la posicion anterior el nodo siguiente a la posición deseada, prescindiendo así del nodo en la posicion deseada, para la eliminación al final simplemente se llega al penúltimo elemento de la lista, se prescinde del nodo siguiente y se le asigna una referencia null pointer, teniendo en ambos casos una complejidad **O(n)**.

  Véase la siguiente imagen.



Cabe recalcar que se describieron los algoritmos mas significativos de la solución, dejando de lado los que resultaban triviales, como los de validación, los cuales son resueltos con una simple comparación.

Problemas de desarrollo

Durante la implementación y el desarrollo de la solución hubo dos problemas significativos, uno de ellos tuvo que ver con el ámbito de los objetos, el problema que aparecía en el programa fue que en muchos de los métodos se definían variables que representaban la instancia actual sea cual fuere la clase, a la hora de que el metodo terminaba de ejecutarse el tiempo de vida de la variable mencionada terminaba, por lo tanto se llamaba a su destructor para liberar la memoria, esto hacía que se liberara memoria de datos los cuales iban a ser utilizados más adelante en la ejecución del programa, por lo tanto hubo infinidad de errores de tipo segmentation fault.

Por otro lado, el segundo problema consistió en la imposibilidad de utilizar contenedores de la biblioteca estándar, lo que sucede es que el programa se habia realizado en su totalidad basándose en contenedores tipo map, ya finalizada la implementación se conoció la restricción en cuanto a los contenedores, por lo tanto se crearon clases orientadas a representar una lista enlazada, dado que se desconocía como implementar un map (árbol binario) de forma manual en C++ se decidió implementar una lista enlazada, lo cual era mucho más familiar que un árbol binario.

Evolución de la solución

A continuación, se dará una breve descripción de los commits realizados en GitHub, con esto se quiere dejar ver la evolución de la solución, en especial la implementación.

- **Inicialización de clases – abril 27**
  En este commit se definieron las clases que inicialmente iban a formar parte de la solución, durante el análisis del problema se dedujo que las clases estación, línea y red iban a ser fundamentales, y a partir de ellas se

desarrollaría la solución, simplemente se definieron los archivos header, nada más.

- **Cuerpo de clases línea, red y estación – abril 28**
  En este momento se definieron los métodos y atributos principales de las clases, a su vez se introdujeron distintos módulos que servirían para abarcar el apartado de experiencia de usuario, tales módulos fueron menus, utilidades y validación, cuya implementación extrajo de la implementación del desafío 1.

- **Menú creado – abril 30**
  El menú de usuario fue creado con la mayoría de sus funcionalidades, a su vez se modificaron algunos de los métodos de las clases tuvieron modificaciones según las necesidades que se iban presentando a lo largo del cumplimiento de cada funcionalidad implementada.

- **Requisitos terminados – mayo 1**
  Se subsanaron todas las funcionalidades que exigía el desafío, en aquel momento hacia falta definir los destructores de las clases, y ubicar mensajes de éxito en las partes del codigo donde el usuario realizara una operación exitosamente, en este punto no se conocía la prohibición en cuanto al uso de contenedores.

- **Actualización – mayo 4**
  En esta etapa del desarrollo de la solución se conoció la prohibición del uso de contenedores, así que se hizo la transición de contenedores map a una lista enlazada implementada de manera manual, sin usar bibliotecas. Dado esto se modifico gran parte de las clases fundamentales (línea y red) para adaptarlas a la nueva lista enlazada, se creo una plantilla que creaba nodos que contenían datos de cualquier tipo, a su vez se creo una lista que contenía tales nodos. Los destructores estaban generando fallos ya que se cometía segmentation fault, para este día no se habia descubierto como solucionar tales problemas sin borrar los destructores.

- **Documentación – mayo 6**
  Se incluyo documentación en gran parte del codigo, a su vez se solucionó el problema del error tipo segmentation fault relacionado a los destructores, para este momento solo faltaba adjuntar el informe, realizar el video y comentar el main.

- **Finalización – mayo 8**
  Se terminó por completo la documentación, y se adjuntó el informe y el link del video al repositorio.

Justificación de la elección de una lista enlazada

Como se pudo ver a lo largo de todo el informe la estructura de datos que se utilizó para almacenar las líneas y estaciones fue una lista enlazada, ante la imposibilidad de utilizar contenedores se decidió usar una lista enlazada ya que era una estructura de datos familiar, además de eso ofrecía una ventaja significativa ante los arreglos convencionales, esto debido a que para utilizar arreglos se debía alojar memoria y copiar el arreglo cada vez que se quisiera editar la dimensión del mismo, no resultaba practico estar realizando copias de los datos constantemente, dado esto se decidió sacrificar el acceso aleatorio y darle prioridad a la inserción, eliminación y la propiedad dinámica de las listas enlazadas (expansión en tiempo de ejecución, sin necesidad de crear nuevas listas), aunque pudo haber resultado más eficiente considerar una estructura de datos no lineal como un grafo o un árbol con el fin de representar las redes metro, tales estructuras no son del todo familiares para el autor, debido a limitaciones de tiempo se decidió implementar una lista simplemente enlazada con atributos para su inicio y final, con su respectivo nodo.