OBLIGATORIO PROGRAMACIÓN III

Santiago Pereira - C.I. 4.618.737-5
Sabine Theiss - C.I. 5.781.191-9

a) Modelado del Grafo:

Será un grafo simple y no dirigido.
Vértice -> Número de las ciudades
Aristas -> Cantidad de recorridos / Indicación si existe una conexión entre 2 ciudades o no

b) Análisis de TAD

Usaremos los siguientes tipos de datos principales:

El conjunto de líneas: **Diccionario**
El conjunto de ciudades y el conjunto de las paradas: **Secuencia**, ambos siendo **secuencias simples**.
La conexión entre ciudades: **Grafo**
Productos cartesianos para agrupar datos de diferentes tipos (ciudad,linea)

Líneas = Diccionario (Línea)
Se elige un diccionario ya que las líneas se identifican con una clave (código de línea), a través de la cual se accede a cada elemento del diccionario.
Es decir, permite tener elementos con un respectivo índice que los ordena. Este índice no permite duplicidad, el mismo estará fijado por el código de la línea.
El diccionario contiene un producto cartesiano (línea), ver a continuación.

Linea = Producto Cartesiano (CodigoLinea, CantidadParadas, Paradas)
Entre sus datos tiene una secuencia, que representa el recorrido de la línea; el primer elemento es el origen y el último es el destino.
Se elige Producto Cartesiano para agrupar información de no necesariamente del mismo tipo.

CodigoLinea = String dinámico
CantidadParadas = Entero

Paradas = Secuencia simple (Ciudad)
Se elige una secuencia simple para almacenar los elementos de forma lineal y ordenada.
Necesitamos acceder a todos los elementos de la secuencia en el listado de las paradas (operación 8), por lo cual descartamos Deque y Queue.

Ciudades = Secuencia(Ciudad)
Se elige una secuencia simple para almacenar los elementos de forma lineal.
Además porque precisamos acceder a todos los elementos de la colección (incluidos los que están en el medio, operación 2).

Ciudad = ProductoCartesiano (NumeroCiudad, NombreCiudad)
Se elige para agrupar información de no necesariamente del mismo tipo.

NumeroCiudad = Entero
NombreCiudad = String dinámico

ConexionCiudades = Grafo(numCiudad)
Se elige este T.A.D. para poder almacenar dos clases de elementos: Los vértices y los aristas. Por un lado, están los vértices de tipo V (que representan las ciudades) y por otro están las aristas de tipo A (que representa si existe una ruta que une a las dos ciudades).
Es el TAD que permite representar si existe una relación entre los mismos elementos.

numCiudad = Entero

c) Estructura de datos elegida

Para la representación del **diccionario de las líneas**, se elige un **ABB**, ya que no hay cota máxima y esa estructura facilita el listado de los elementos de forma ordenada (se pide en el punto 6). Además, no puede haber elementos repetidos.
Por la forma que están ordenados estos elementos, el orden de las funciones de búsqueda es menor frente a otras estructuras de datos, la cual se pide en el requerimiento 7 y 8.

Se elige una **LPPF** para la **Secuencia de las paradas**, ya que no existe cota máxima de paradas por línea. La LPPF facilita el ingreso de nuevos elementos al final (la parada destino), y acceder al primer elemento (parada de comienzo). Además puede haber paradas repetidas, lo cual permite este estructura.
La LPPF, ofrece la función de insertar elementos al final con un Orden (1), siendo más eficiente el ingreso de nuevos elementos (la parada destino), y acceder al primer elemento (parada de comienzo) también es de Orden (1).

Para la **secuencia de ciudades**, se opta por usar un **Arreglo con tope**, ya que el número de ciudades (N) es fijo y definido de antemano.
Además, los números de las ciudades irán siendo asignados en forma consecutiva, pudiendo utilizar el tope para designar el número de la ciudad que estamos ingresando en el arreglo.

Para la representación del **Grafo de la conexión entre ciudades** se utilizará una **matriz de adyacencia**, ya que se espera que la empresa registre un alto volumen de tramos, por ende siendo más eficiente la estructura de matriz de adyacencia en comparación con una Lista de adyacencia. En otras palabras, como esperamos tener muchos tramos cubiertos por la empresa, se espera que existan muchas aristas; dado que la función “Hay arista” en la lista de Adyacencia es de Orden (n) comparado con la matriz de Adyacencia que tiene Orden (1), es más eficiente esta última.

Para los **Productos cartesiano**s (**Línea**, **Ciudad**) se utilizan estructuras de “**struct**” para agrupar datos de diferente tipo.

Para el tipo **String** (CodigoLinea, NombreCiudad) se utiliza un **String dinámico** para optimizar el espacio de memoria.

Tipos de datos en C++

Typedefs:

Líneas:

```
typedef struct nodoA
        {
                Linea info;
                nodoA * hizq;
                nodoA * hder;
        } nodo;

typedef nodo * Líneas;
```

Linea:

```
typedef struct
        {
                String CodigoLinea;
                int CantidadParadas;
                Paradas paradas;
        } Linea;
```

Secuencia Paradas:

```
typedef struct nodoL
        {
                Ciudad info;
                nodoL *sig;
        } Nodo;

typedef struct {
                Nodo * prim;
                Nodo * ult;
                } Paradas;
```

Secuencia Ciudades:

```
const int CANT = 20;
typedef struct
        {
                Ciudad arre [CANT];
                int tope;
        } Ciudades;
```

Ciudad:

```
typedef struct
	{
		int NumeroCiudad;
		String NombreCiudad;
	}Ciudad;
```

Grafo Conexión Ciudades:

```
const int CANT_CIUD = 5;
typedef int Grafo[CANT_CIUD][CANT_CIUD];
```

d) Diagrama de módulos