ENTREGA OBLIGATORIO PROGRAMACIÓN III

Viernes 26 de Julio del 2019

Integrantes:
Fabián Negrín - CI 4.908.980-7
Alejandro Ormazábal - CI 4.813.685-1
German Pequera - CI 4.503.242-4

Índice

Modelado en términos de grafo 3
Análisis de Tipos Abstractos de Datos 4
Estructuras de datos para los T.A.D 4
Encabezados de las primitivas y otras operaciones de cada uno de los T.A.D 5
Esquema de módulos e inclusiones 10

Modelado en términos de grafo

Parte de la realidad de la empresa se modelo en términos de grafos, donde:

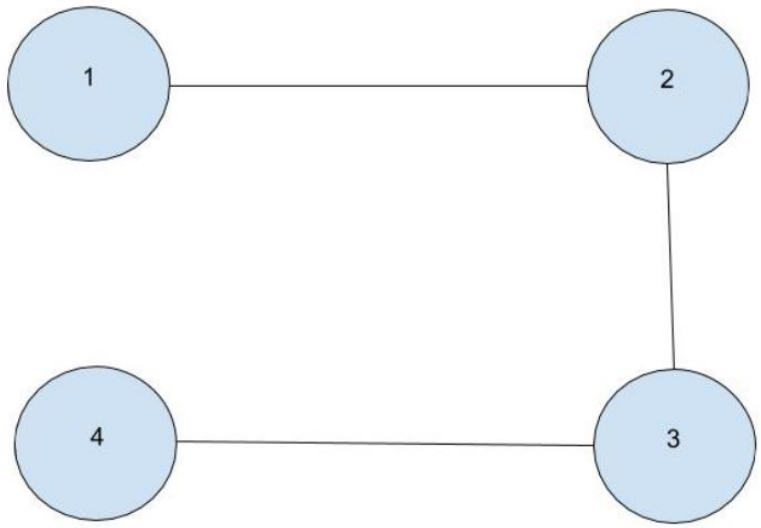
Vértices: representan las paradas(ciudad)
Aristas: representan si hay tramo(camino) que los une.

Los vértices del grafo representan las ciudades (identificadas por un código numérico), y las aristas representan los tramos que unen dos ciudades.

Con esta representación somos capaces de representar la realidad de manera tal, que se pueden especificar diferentes tramos entre las distintas ciudades, lo que a la postre será necesario para determinar diferentes recorridos para las líneas de la empresa.

Este grafo es un grafo no dirigido, simple y dependiendo de los datos y tramos que maneje la empresa podrá ser conexo/disconexo.

Representación Grafo:



1 = Montevideo
2 = Canelones
3 = San José
4 = Colonia

Análisis de Tipos Abstractos de Datos

Ciudades = Diccionario(Ciudad)
Ciudad = Producto Cartesiano (codigoCiudad, nombreCiudad)

Lineas = Diccionario(linea)
Linea = Producto Cartesiano(numeroLinea,Recorrido)

Recorrido = Secuencia(Parada)
Parada = Producto Cartesiano(numeroParada,Ciudad)

codigoCiudad = Entero
numeroLinea = String
numeroParada = Entero
nombreCiudad = String

Diccionarios: elegimos trabajar con diccionarios para las ciudades y para las líneas, ya que ambas se identifican de manera única mediante una clave, donde para las ciudades es el nombre de las mismas y para las líneas es un código alfanumérico.

Para el recorrido trabajaremos con una secuencia ya que necesitamos almacenar sus paradas de manera lineal y además esta nos permite acceder a cualquier para del recorrido.

Estructuras de datos para los T.A.D

GRAFO:
Se eligió matriz de adyacencia ya que en los requerimientos 2 y 3 las operaciones de quedan de orden 1, donde las mismas con una lista de adyacencia serian de orden n por lo cual la 1er opción es mucho más eficiente y preferible.
Otro motivo para trabajar con matriz de adyacencia, es que la cantidad de tramos entre las ciudades pueden crecer a medida que crece la empresa, lo que implicara que se tienda a trabajar con numero de aristas considerables en comparación con los vértices, y de esta manera el gasto de memoria para representar aristas que no existen, tiene a ser bajo.

Diccionario de Ciudades:
Se eligió estructura de hash, ya que las mimas se identifican de manera única por su nombre y, además, conocemos de antemano el número total de ciudades lo cual nos va a permitir establecer el número adecuado de cubetas, para luego con una buena función de dispersión poder acceder eficientemente a cada una de ellas.

Secuencia de Recorridos:
Se eligió una estructura de LPPF ya que para poder agregar una nueva parada al final del recorrido de una nueva línea. Este es un requerimiento solicitado.

Otro requerimiento pide listar la información de las líneas, y dentro de la información solicitada pide mostrar específicamente su origen y destino, es por eso también que el puntero al inicio y al final, implica resolver de manera eficiente el requerimiento.
Por otra parte, haber agregado un doble enlace o hacer la lista circular, no aporta ningún beneficio para los requerimientos solicitados, e incluso por sus características en algún caso podría ser hasta perjudicial en términos de eficiencia.

Diccionario de Líneas:
Se eligió estructura de ABB ya que uno de los requerimientos pide listar ordenado alfanuméricamente, con lo cual entendemos que una estructura de árbol es más adecuada que una estructura de hash, ya que iremos almacenando las líneas alfabéticamente, haciendo posible este requerimiento se realice de manera eficiente.

Productos Cartesiano:
CIUDAD, LINEA, PARADA:
En los tres casos elegimos un tipo de datos estructurados, ya que para los tres nos interesa almacenar características que no necesariamente son del mismo tipo.

Encabezados de las primitivas y otras operaciones de cada uno de los T.A.D

MÓDULO STRING:

void strcrear(String &s);

void strdestruir(String &s);

int strlar(String s);

void strcop(String &s1, String s2);

.void scan(String &s);

void print(String s);

boolean strmen(String s1, String s2);

boolean streq(String s1, String s2);

//devuelve la suma de los caracteres de un string
int SumaAscii(String nombreCiudad);

//pasa todo el string a mayúscula
void StringAMayusculas(String &s);

MÓDULO GRAFO:

///Crea el Grafo(Matriz) vacio
void CrearGrafo(Grafo &G);

///Retorna si existe la arista, en este contexto retorna si existe un tramo entre las ciudades
boolean ExisteTramo(Grafo G, int codigoCiudad1, int codigoCiudad2);

////(InsertarTramo) = Inserta la nueva arista al grafo. En este contexto se inserta un nuevo tramo entre dos ciudades.
void InsertarArista(Grafo &G, int codigoCiuadad1, int codigoCiuadad2);
///Precondición: la arista no pertenece al grafo. -> !ExisteTramo

/// Retorna si existe alguna secuencia de tramos entre dos ciudades; si existe un camino entre dos vertices
boolean ExisteSecuenciaDeTramoEntreDosCiudades(Grafo G, int codigoCiudad, int codigoCiudad2);

/// Ejecuta DFS para saber si existe un camino entre dos vertices(ciudades)
void DFS(Grafo G, int verticeActual, int destino, boolean visitado[CANT_CIUDADES]);

/// Retorna TRUE si existe al menos una atista en el grafo.
/// en este contexto quiere decir que existe al menos un tramo definido
boolean ExiteAlMenosUnTramo(Grafo g);

MÓDULO CIUDAD:

//Retorna un string con el nombre de la ciudad
void DarNombre(Ciudad c, String &nombre);

/// Retorna un entero con en código de la ciudad
int DarCodigo(Ciudad c);

/// Muestra en pantalla los datos de la ciudad
void MostrarCiudad(Ciudad c);

///Solicita y carga los datos necesarios de una ciudad
void CargarCiudad(Ciudad &c, int codigoCiudad);

MÓDULO CIUDADES:

/// Crea el Hash de ciudades vacío
void MakeCiudades(Ciudades &C);

///Determina si en el diccionario existe la ciudad especificada por su clave (nombre)
boolean Member(Ciudades C, String nombreCiudad);

///Inserta la ciudad en el Hash
void Insert(Ciudades &C, Ciudad ciu);
/*Precondición: la ciudad no existe en el Hash -> !Member(nomCiudad)*/

/// Retorna la ciudad correspondiente a la clave(nombre)
Ciudad FindCiudad(Ciudades C, String nomCiudad);
///Precondición: la ciudad es existe en el Hash -> member(nomCiudad)

MÓDULO LÍNEA:

/// Retorna un String con el código de la línea
void DarCodigoLinea(Linea l,String &nomLinea);

///Retorna el recorrido de la línea
Recorrido DarRecorrido(Linea l);

///Muestra en pantalla la información correspondiente a la linea
void MostrarLinea(Linea l);

/// Solicita y cargar la información necesaria de una linea
void CargarLinea(Linea &l);

/// Agrega la parada al recorrido de linea
void AgregarParadaARecorridoDeLinea(Linea &l, Parada p);
///Precondición: Sí recorrido !RecorridoVacio exitetramo(DestinoRecorido,nuevaParada)

/// Lista la informacion de todas las paradas del recorrido(número de parada y nombre de ciudad)
void ListarParadasDeRecorridoEnLinea(Linea l);

MÓDULO LÍNEAS:

/// Crea el ABB de líneas vacío
void MakeLineas(Lineas &l);

/// Retorna TRUE si en el ABB existe una línea con la clave especificada(codigoLinea)
boolean MemberLinea(Lineas l,String codigoLinea);

/// Inserta la línea en el ABB
void InsertLinea(Lineas &lineas, Linea l);
/*Precondición: La linea no existe en el ABB -> !MemberLinea(codigoLinea)*/

/// Retorna la línea con la clave especificada(codigoLinea)
Linea FindLinea(Lineas l, String codigoLinea);
///Precondición: La línea existe en el ABB -> MemberLinea(codigoLinea)

/// Sustituye la vieja línea en el ABB por la nueva recibida por parámetro
void ModifyLinea(Lineas &l,Linea linea);
///Precondición: La linea a sustituir es miembro del ABB. -> Member(codigoLinea)

/// Listar los datos básicos de todas las líneas de la empresa (código, origen, destino y
/// cantidad de paradas), ordenados por código alfanumérico de menor a mayor.
void ListarLineas(Lineas l);

/// Retorna TRUE si el ABB es vacío
boolean lineasVacia(Lineas l);

MÓDULO PARADA:

/// Retorna un entero con el numero de la parada
int DarNumero(Parada p);

/// Retorna la ciudad de la parada
Ciudad DarCiudad(Parada p);

/// Muestra por pantalla los datos de la parada
void MostrarParada(Parada p);

/// Muestra por pantalla los datos de la parada, se muestra con otro formato
void MostrarParadaAlternativo(Parada p);

/// Carga los datos correspondientes a la parada
void CargarParada(Parada &p, int numParada, Ciudad ciudParada);

MÓDULO RECORRIDO:

/// Crea la lista de Recorrido vacía
void CrearRecorrido(Recorrido &r);

/// Inserta una parada al final de la lista
void InsBackRecorrido(Recorrido &r, Parada p);

/// Retorna TRUE si el recorrido(lista) es vacio

boolean RecorridoVacio(Recorrido &r);

/// Retorna un entero con la la cantidad de elementos(paradas) del recorrido
int LargoRecorrido(Recorrido r);

/// Muestra las paradas del recorrido
void MostrarRecorrido(Recorrido r);

/// Muestra la primer parada del recorrido
void MostrarOrigen(Recorrido r);
/* Precondición: el recorrido no es vacío -> !RecorridoVacio(r) */

/// Muestra la última parada del recorrido
void MostrarDestino(Recorrido r);
/* Precondición: el recorrido no es vacío -> !RecorridoVacio(r) */

/// Retorna la última parada del recorrido
Parada Destino (Recorrido r);
/* Precondición: el recorrido no es vacío -> !RecorridoVacio(r) */

MÓDULO MENU:

void MostrarMenu(int &opcion);

Esquema de módulos e inclusiones