Estructuras de Datos

Trabajo Práctico

Segundo semestre 2022

Presentación

En este trabajo práctico vamos a trabajar con el tipo abstracto de datos **`DualNet`**, similar al trabajado en el parcial, pero en C/C++. Las redes de conexión manejadas por una empresa se representarán a través del TAD **`DualNet`**, con la interfaz provista en el archivo **`DualNet.h`**. La representación para implementarlo utiliza:

- un TAD **`Switch`** que lleva cuenta de las conexiones de la empresa, y
- un **`Map`** que relaciona a cada cliente conectado con la ruta de conexión.

El tipo abstracto **`Switch`** guarda todas las rutas de conexión de la empresa, y su interfaz se provee en el archivo **`Switch.h`**. Para implementar se proveen una serie de módulos ya programados, y las interfaces de aquellos que deben programar. El trabajo a realizar es completar la implementación provista, según se describe en las secciones siguientes. En cada caso se explican los módulos que existen y los que deben programar, con los detalles correspondientes. Para mayores referencias deben recurrir al código provisto.

Módulos existentes

Los módulos existentes proveen los tipos básicos, incluidas listas varias y un map. Dado que las estructuras de datos en C no son polimórficas, se repiten las implementaciones para diferentes listas con diferentes nombres de operaciones. **El código de estos módulos no se debe modificar**.

Los módulos existentes son:

- **`Cliente`**, implementa el tipo **`Cliente`** como una abstracción de strings con operaciones de tamaño y comparación.
- **`Clientes`**, implementa listas de clientes, con operaciones para agregar, mostrar y recorrer estas listas.
- **`Ruta`**, implementa las rutas como listas de bocas, con operaciones para agregar al comienzo y al final de una ruta, copiar y mostrar una ruta, y recorrerla.
- **`Rutas`**, implementa listas de rutas, con operaciones para agregar, mostrar y recorrer estas listas, y también algunas operaciones más específicas necesarias para la implementación de **`Switch`**.
- **`MapCR`**, implementa un **`Map`** que asocia un **`Cliente`** con una **`Ruta`**, equivalente al tipo de Haskell **`Map Cliente Ruta`**. La interfaz es similar a la vista para el TAD de Haskell, con las modificaciones correspondientes por trabajar en un lenguaje imperativo, más dos operaciones de muestra de información. Está implementado con un BST basado en punteros, y se recomienda analizar la implementación para aprender los detalles. Las dos operaciones para realizar la muestra de información trabajan una en forma abstracta, que se espera sea la que usen los usuarios, y otra que evidencia la estructura de BST, útil para verificar la implementación durante el proceso de codificación.

Las interfaces específicas de cada módulo pueden consultarse en el código adjunto, en los archivos de extensión **`.h`** con los nombres correspondientes. En el caso de las listas se proveen operaciones de recorrido mediante iteradores, para usar según lo visto en las clases.

Los tipos de Haskell que estos módulos proveen, adaptados al lenguaje C/C++, son los siguientes:

```
type Cliente = String                      -- En el módulo Cliente
data Boca    = Boca1 | Boca2 deriving Eq   -- En el módulo Ruta
type Ruta    = [Boca]                      -- En el módulo Ruta
```

```
type Clientes = [Cliente]                    -- En el módulo Clientes
type Rutas    = [Ruta]                       -- En el módulo Rutas
type MapCR = Map Cliente Ruta                -- En el módulo MapCR
```

Además existen algunos módulos que fueron utilizados para verificar las implementaciones. Estos módulos pueden modificarse libremente para probar otras características deseadas, pero se espera que al menos funcionen correctamente con las pruebas provistas.

Módulos a codificar

Los módulos a codificar ya contienen algo de código ya provisto, para guiar en la implementación. En particular se definen las interfaces, partes de las estructuras y en cada caso se proveen operaciones que permiten mostrar la estructura por consola, para poder controlar y realizar testings visualmente. El código ya provisto **no debe modificarse**. Las operaciones que deben programar aparecen con un comentario que indica que deben completarse.

BinHeapC

Este módulo provee una **`Heap`** que guarda pares **`(Int, Cliente)`**, equivalente al tipo de Haskell **`Heap (Int, Cliente)`**. La interfaz es similar a la vista para el TAD de Haskell, con las modificaciones correspondientes por trabajar en un lenguaje imperativo.

```
type BinHeapC = Heap (Int, Cliente)          -- En el módulo BinHeapC
```

El código para la implementación ya provee los tipos necesarios para implementar esta heap usando una heap binaria basada en arrays, como vimos en la clase teórica, y las operaciones para mostrar una estructura de heap.

Observar que existen DOS operaciones para obtener el mínimo: una para el entero y otra para el cliente. Esto se realiza así para no tener la necesidad de definir un tipo para los pares. Se espera que ambas operaciones sean de O(1), por lo que el costo extra no es excesivo con respecto a la ganancia en memoria.

Switch

Este módulo provee un **`Switch`** que asocia un **`Cliente`** con una **`Ruta`**. La interfaz es similar a la vista para el TAD de Haskell, con las modificaciones correspondientes por trabajar en un lenguaje imperativo.

```
data Switch a = Terminal
              | Conmutador (RedPrivada a) (Switch a) (Switch a)
data RedPrivada a = Disponible
                  | Conexion a
```

El código para la implementación ya provee los tipos necesarios para implementar esta estructura usando un árbol binario, y las operaciones para mostrar la estructura según las rutas disponibles. Para el tipo que en Haskell se implementa como **`RedPrivada`**, se utilizará el mismo tipo **`Cliente`**, con el valor nulo (inválido según el TAD **`Cliente`**), para indicar una conexión disponible, en forma similar a como se realizó para el resultado de tipo **`Maybe`** en el **`lookupMCR`**. También el caso **`Terminal`** de Haskell en el **`Switch`** se representa con el puntero nulo.

La operación de **`Conectar`** debe considerar la ruta dada para conectar al cliente en la posición adecuada de la estructura, teniendo en cuenta que en algunos casos se deberán crear nuevos nodos para poder actualizar la posición adecuada.

La operación de **`Desconectar`** debe encontrar la posición indicada por la ruta, teniendo en cuenta que puede suceder que la misma no exista, en cuyo caso no deberá hacer nada más. Como bonus no exigido

puede intentarse satisfacer el invariante de representación de que ningún nodo tiene sus tres campos en nulo (en cuyo caso puede reemplazarse por un puntero nulo directamente), pero esto es más complejo de lo que se espera que puedan hacer en general.

La operación **`disponiblesADistancia`** debe considerar el caso de que el árbol se represente con un puntero nulo, pero continuar incorporando los valores adecuados a las rutas a buscar hasta alcanzar la longitud buscada.

DualNet

Este módulo provee un TAD para redes de conexión que llevan conexiones de **`Cliente`**s a la red. La interfaz es similar a la vista para el TAD de Haskell, con las modificaciones correspondientes por trabajar en un lenguaje imperativo.

```
data Dualnet = DN (Switch Cliente)    -- Indica todas las conexiones en curso
                  (Map Cliente Ruta)  -- Asocia cada cliente con su ruta
```

Se proveen las declaraciones de registro necesarias para obtener esta representación, con el invariante de representación esperado, y las operaciones a proveer. También se provee una operación para mostrar el valor de una red por consola, con fines de control durante el trabajo.

Entrega del trabajo práctico

Deben entregarse los archivos **`BinHeapC.cpp`**, **`Switch.cpp`** y **`DualNet.cpp`** de forma tal que al ser utilizados con el resto de los archivos provistos se ejecuten de la forma esperada. Para ello deben enviar un correo a tpi-doc-ed@listas.unq.edu.ar con Tema **“Entrega TP EstrD de *<Apellido, Nombre>*”** (reemplazando ***<Apellido, Nombre>*** con su propio apellido y nombre), y conteniendo un archivo adjunto de nombre **`TP-EstrD-2022s2-Apellido-Nombre.zip`** (donde nuevamente **`Apellido-Nombre`** fue reemplazado por los datos correspondientes). Este archivo también debe ser subido al aula virtual. El archivo debe contener *solamente* los 3 archivos mencionados.

El criterio de corrección incluye código que NO se cuelga, que ofrece la salida esperada, y que sigue los principios de trabajo transmitidos durante la cursada. Se probarán los 3 archivos enviados en una implementación idéntica a la suministrada, pero posiblemente con diferentes pruebas en los módulos de test.

Se puede entregar hasta dos veces. La primera entrega es como máximo el 18/11, y la entrega recuperatorio es como máximo el 25/11. Aquellas entregas que se reciban antes del 18/11 recibirán un feedback y pueden ser corregidas y reentregadas hasta la entrega recuperatorio. Aquellas entregas que se reciban entre el 19/11 y el 25/11 NO tendrán oportunidad de reentrega, y en caso de no ser correctas desaprobarán. Quiénes no hayan entregado para el 25/11 se considerarán ausentes. Todas las entregas aprobadas deben ser defendidas el 7/12.