Trabajo práctico especial - Primera parte

30 de mayo de 2016

Integrantes

Nombre y apellido: Abel Matías Osorio.

Correo electrónico: abel.m.osorio@gmail.com.

Número de grupo: 60.

Ayudante: Rubi, Pablo.

Especificación formal en Nereus y estructuras de datos

TDA "Lista"

```
CLASS Fila [Elemento]
  IMPORTS Nat
  BASIC CONSTRUCTORS inicLista, agregarPrincipioLista, agregarFinalLista, agregarLista
  EFFECTIVE     TYPE Lista
    OPERATIONS        inicLista: -> Lista;
      longLista: Lista -> Nat;
      agregarPrincipioLista: Lista * Elemento -> Lista;
      agregarFinalLista: Lista * Elemento -> Lista;
      agregarLista: Lista(l) * Nat(i) * Elemento (e) -> Lista
        pre: (i >= 1) and (i <= longLista(l));
      estaVacia: Lista -> Boolean;
      eliminarLista: Lista(l) * Nat(i) -> Lista
        pre: (i >= 1) and (i <= longLista(l)) and not estaVacia(l);
      recuperarLista: Lista(l) * Nat(i) -> Elemento
        pre: (i >= 1) and (i <= longLista(l)) and not estaVacia(l);
      estaIncluido: Lista(l) * Elemento -> Boolean
        pre: not estaVacia(l);
      leerPrimero: Lista(l) -> Elemento
        pre: not estaVacia(l);
      leerSiguiente: Lista(l) -> Elemento
        pre: not estaVacia(l);
END-CLASS
```

Estructura de datos utilizada

Para implementar el TDA *Lista* se utilizó un struct *Nodo* con los siguientes atributos:

elemento Almacena el valor del nodo.

siguiente Puntero al siguiente nodo de la lista.

Esta estructura permite una fácil implemenación de la lista, y además permite poder agregar elementos de manera flexible sin ningún límite de tamaño. Además, se utiliza sólo la memoria necesaria para almacenar los elementos de la lista, es decir, no es necesario reservar memoria para futuros posibles elementos. Como punto en contra, no se tiene una forma directa de acceder a un elemento arbitrario de la lista, sino que hay que recorrer la misma hasta llegar al elemento buscado. Con una estructura como, por ejemplo, un arreglo, se podría mejorar el acceso a nodos arbitrarios de la lista y determinar fácilmente cuál es el primer y último nodo, entre otras cosas. Sin embargo, sería necesario reservar, al momento de inicializar la lista, un bloque de memoria contiguo lo suficientemente grande como para poder generar las listas que se quieran.

TDA "Arbol"

```
CLASS Arbin [Elemento]
  IMPORTS Nat, Lista
  BASIC CONSTRUCTORS inicArbin, agregarElemento
  EFFECTIVE
```

```
    TYPE Arbin
    OPERATIONS
      inicArbin: -> Arbin;
      vacioArbin: Arbin -> Boolean;
      recuperarRaiz: Arbin(t) -> Elemento
        pre: not vacioArbin(t);
      agregarElemento: Arbin * Elemento -> Arbin;
      listarInorden: Arbin(t) -> Lista
        pre: not vacioArbin(t);
      estaIncluido: Arbin(t) * Elemento -> Boolean
        pre: not vacioArbin(t);
      cantidadNodos: Arbin -> Nat;
      profundidad: Arbin -> Nat;
      construirFrontera: Arbin(t) -> Lista
        pre: not vacioArbin(t);
      esHoja: Arbin(t) * Elemento -> Boolean
        pre: not vacioArbin(t);
END-CLASS
```

Estructura de datos utilizada

Para implementar el TDA *Arbin* se utilizó una estructura recursiva de structs *Nodo* con los siguientes atributos:

elemento Almacena el valor del nodo.

subIzquierdo Puntero al nodo izquierdo.

subDerecho Puntero al nodo derecho.

Todos los valores que estén almacenados en la rama izquierda serán menores a la raíz, y todos los almacenados a la derecha mayores. La ventaja de utilizar esta estructura es que permite recorrer el árbol de manera recursiva obteniendo, en general, complejidades logarítmicas, y a su vez el espacio que ocupa en memoria es prácticamente sólo el que necesita para almacenar los valores contenidos, sin necesidad de sobredimensionar ninguna estructura.

Código fuente

Implementación del TDA *Lista*

lista.h

```
#ifndef LISTA_H_INCLUDED
#define LISTA_H_INCLUDED

#include <iostream>

using namespace std;

template <typename Elem>
class Lista {
    private:
        struct Nodo {
            Elem elemento;
            Nodo * siguiente;
        };
        // Inicio de la lista.
        Nodo * inicio;
        // Puntero para recorrer la lista. Indica la posición del elemento
        // apuntado. Comienza en 1.
        int punteroPosicion;
```

```
        // Puntero para recorrer la lista. Es el puntero real del elemento
        // apuntado.
        Nodo * puntero = NULL;
        // Función para vaciar la lista.
        void vaciar();

    public:
        // Inicializadora de la lista.
        // Complejidad: O(1).
        Lista();
        // Destructora de la lista.
        // Complejidad: O(n).
        ~Lista();
        // Devuelve el tamaño de la lista (cantidad de elementos incluídos).
        // Complejidad: O(n).
        int longLista() const;
        // Agrega un elemento al principio de la lista.
        // Complejidad: O(1).
        void agregarPrincipioLista(Elem);
        // Agrega un elemento al final de la lista.
        // Complejidad: O(n).
        void agregarFinalLista(Elem);
        // Agrega un elemento en una posición de determinada de la lista.
        // Complejidad: O(n).
        void agregarLista(int, Elem);
        // Elimina un elemento de la lista.
        // Complejidad: O(n).
        void eliminarLista(int);
        // Devuelve el elemento de una posición determinada.
        // Complejidad: O(n).
        void recuperarLista(int, Elem &) const;
        // Indica si un elemento está incluído en la lista.
        // Complejidad: O(n).
        bool estaIncluido(Elem) const;
        // Función para chequear si la lista está vacía.
        // Complejidad: O(1).
        bool estaVacia() const;
        // Función para mostrar el valor actual de la lista. El valor es
        // almacenado en el parámetro que pasa por referencia.
        // Complejidad: O(1).
        void leerPrimero(Elem \I&);
        // Función para obtener el próximo valor de la lista. El puntero
        // se corre a la próxima posición. El valor es almacenado en el
        // parámetro que pasa por referencia.
        // Complejidad: O(1).
        void leerSiguiente(Elem &); };
#endif
```

lista.cpp

```
#include "lista.h"

/* Constructora del objeto.
 * Inicializa el primer nodo con valores NULL.
 */
template <typename Elem>
Lista<Elem>::Lista()
```

```
{
    inicio = NULL;
    punteroPosicion = 1;
}

/* Destructora del objeto.
 * Llama a vaciar() para eliminar todos los nodos de la lista.
 */
template <typename Elem>
Lista<Elem>::~Lista()
{
    vaciar();
}

/* Función para vaciar la lista.
 */
template <typename Elem>
void Lista<Elem>::vaciar()
{
    Nodo * aux;
    while (inicio->siguiente != NULL) {
        aux = inicio->siguiente->siguiente;
        delete inicio->siguiente;
        inicio->siguiente = aux;
    }

    inicio->siguiente = NULL;
}

/* Función para agregar un elemento al principio de la lista.
 */
template <typename Elem>
void Lista<Elem>::agregarPrincipioLista(Elem valor)
{
    Nodo * nuevo;

    nuevo = new Nodo;
    nuevo->siguiente = inicio;
    nuevo->elemento = valor;

    inicio = nuevo;
}

/* Función para agregar un elemento al final de la lista.
 */
template <typename Elem>
void Lista<Elem>::agregarFinalLista(Elem valor)
{
    Nodo * aux = inicio;
    Nodo * nuevo;

    if (aux != NULL) {
        while (aux->siguiente != NULL) {
            aux = aux->siguiente;
        }
        nuevo = new Nodo;
```

```
        nuevo->siguiente = NULL;
        nuevo->elemento = valor;

        aux->siguiente = nuevo;
    } else {
        // La lista está vacía. Agregar el elemento al inicio o al final de
        // la lista, es lo mismo.
        agregarPrincipioLista(valor);
    }
}

/* Función para agregar un elemento en una posición arbitraria de la lista.
 */
template <typename Elem>
void Lista<Elem>::agregarLista(int posicion, Elem valor)
{
    if ((posicion < 1) or (posicion > longLista()) or estaVacia()) {
        return;
    }

    Nodo * aux = inicio;
    Nodo * nuevo;
    int actual = 0;

    if (aux != NULL) {
        while (aux->siguiente != NULL and posicion < actual) {
            aux = aux->siguiente;
        }
        nuevo = new Nodo;
        nuevo->siguiente = aux->siguiente;
        nuevo->elemento = valor;

        aux = nuevo;
    } else {
        // La lista está vacía. Se agrega el elemento al principio.
        agregarPrincipioLista(valor);
    }
}

/* Devuelve el tamaño de la lista (cantidad de elementos incluídos).
 */
template <typename Elem>
int Lista<Elem>::longLista() const
{
    Nodo * aux = inicio;
    int longitud = 0;

    while (aux != NULL) {
        longitud++;
        aux = aux->siguiente;
    }

    return longitud;
}

/* Función para determinar si un elemento está incluído en la lista.
```

```
 */
template <typename Elem>
bool Lista<Elem>::estaIncluido(Elem buscado) const
{
    if (estaVacia()) {
        return false;
    }

    Nodo * aux = inicio;
    while (aux != NULL) {
        if (aux->elemento == buscado) {
            return true;
        }
        aux = aux->siguiente;
    }

    return false;
}

/* Función para chequear si la lista está vacía.
 */
template <typename Elem>
bool Lista<Elem>::estaVacia() const {
    return (inicio == NULL);
}

/* Función para eliminar un elemento de una posición determinada.
 */
template <typename Elem>
void Lista<Elem>::eliminarLista(int posicion)
{
    // Si la posición es menor a 1 o excede la longitud de la lista, entonces
    // no hago nada.
    if ((posicion < 1) or (posicion > longLista()) or estaVacia()) {
        return;
    }

    int actual = 1;
    Nodo * aux = inicio;
    Nodo * victima;
    Nodo * anterior = NULL;

    while (actual < posicion) {
        anterior = aux;
        aux = aux->siguiente;
        actual++;
    }

    victima = aux;

    if (anterior != NULL) {
        anterior->siguiente = aux->siguiente;
    } else {
        // Se está borrando el primer nodo, por lo tanto hay que
        // actualizar el puntero de inicio de la lista.
        inicio = aux->siguiente;
```

```
    }

    delete victima;
}

/* Función para obtener un valor de la lista.
 */
template <typename Elem>
void Lista<Elem>::recuperarLista(int posicion, Elem & valor) const
{
    if ((posicion < 1) or (posicion > longLista()) or estaVacia()) {
        return;
    }

    int actual = 1;
    Nodo * aux = inicio;

    while (actual < posicion) {
        actual++;
        aux = aux->siguiente;
    }

    valor = aux->elemento;
}

/* Función para obtener el primer valor de la lista.
 */
template <typename Elem>
void Lista<Elem>::leerPrimero(Elem & primero)
{
    if (!estaVacia()) {
        primero = inicio->elemento;
        punteroPosicion = 2;
        puntero = inicio->siguiente;
    }
}

/* Función para obtener el próximo valor de la lista.
 */
template <typename Elem>
void Lista<Elem>::leerSiguiente(Elem & siguiente)
{
    if (!estaVacia()) {
        if (punteroPosicion <= longLista()) {
            siguiente = puntero->elemento;
            puntero = puntero->siguiente;
            punteroPosicion++;
        }
    }
}

// Tipos para los cuales la clase está implementada
template class Lista<unsigned int>;
template class Lista<int>;
template class Lista<float>;
template class Lista<char>;
```

```
template class Lista<string>;
```

Implementación del TDA Arbin

arbin.h

```
#ifndef ARBOL_BINARIO_H_INCLUDED
#define ARBOL_BINARIO_H_INCLUDED

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

using namespace std;

template <typename Elem>
class Arbin {
    private:
        struct Nodo {
            Elem elemento;
            Nodo * subIzquierdo;
            Nodo * subDerecho;
        };

        // Nodo raíz del árbol binario.
        Nodo * raiz = NULL;
        // Función privada para eliminar todos los nodos del árbol.
        void vaciar(Nodo *);
        // Función privada para agregar un elemento al árbol.
        void agregarElemento(Nodo *, Elem);
        // Función privada para lista los elementos del árbol inorden.
        void listarInorden(Nodo *, Lista<Elem> * &) const;
        // Función privada para determinar si un elemento está incluído en el
        // árbol.
        void estaIncluido(Nodo *, Elem, bool &) const;
        // Función privada para contar la cantidad de elementos del árbol.
        int cantidadNodos(Nodo *) const;
        // Función privada para determinar la profundidad del árbol.
        int profundidad(Nodo *) const;
        // Función para determinar si un nodo es hoja.
        // Complejidad: O(1).
        bool esHoja(Nodo *) const;
        // Función privada para construir la frontera del árbol.
        void construirFrontera(Nodo *, Lista<Elem> * &) const;

    public:
        // Función constructora del árbol binario.
        // Complejidad: O(1).
        Arbin();
        // Función destructora del árbol.
        // Complejidad: O(n).
        ~Arbin();
        // Función para determinar si el árbol está vacío.
        // Complejidad: O(1).
        bool vacioArbin() const;
        // Función para obtener el elemento raíz del árbol.
        // El valor de la raíz se almacenará en el parámetro enviado.
        // Complejidad: O(1).
        void recuperarRaiz(Elem &) const;
```

```
        // Función para agregar un elemento al árbol.
        // Complejidad: O(log(n)).
        void agregarElemento(Elem);
        // Función para listar los elementos del árbol inorden.
        // Los elementos se agregarán en la lista pasada como parámetro.
        // Complejidad: O(n).
        void listarInorden(Lista<Elem> * &) const;
        // Función para determinar si un elemento está incluído en el árbol.
        // Complejidad: O(log(n)).
        bool estaIncluido(Elem) const;
        // Función para contar la cantidad de nodos que tiene el árbol.
        // Complejidad: O(n).
        int cantidadNodos() const;
        // Función para determinar la profundidad del árbol.
        // Complejidad: O(n).
        int profundidad() const;
        // Función para construir la frontera del árbol.
        // Los elementos de la frontera se agregarán en la lista pasada como
        // parámetro.
        // Complejidad: O(n).
        void construirFrontera(Lista<Elem> * &) const;
};

#endif
```

arbin.cpp

```
#include "arbin.h"

/* Inicializadora del árbol binario.
 */
template <typename Elem>
Arbin<Elem>::Arbin()
{
    raiz = NULL;
}

/* Destructora del árbol binario.
 * Ejecuta vaciar() para eliminar uno a uno los nodos del árbol.
 */
template <typename Elem>
Arbin<Elem>::~Arbin()
{
    vaciar(raiz);
}

/* Función para eliminar, uno a uno, todos los nodos del árbol binario.
 */
template <typename Elem>
void Arbin<Elem>::vaciar(Nodo * arbol)
{
    if (arbol != NULL) {
        vaciar(arbol->subIzquierdo);
        vaciar(arbol->subDerecho);
        delete arbol;
    }
}
```

```
/* Función para determinar si el árbol binario está vacío.
 */
template <typename Elem>
bool Arbin<Elem>::vacioArbin() const
{
    return (raiz == NULL);
}

/* Función para obtener el elemento raíz del árbol (si no está vacío).
 */
template <typename Elem>
void Arbin<Elem>::recuperarRaiz(Elem I\& valor) const
{
    if (!vacioArbin()) {
        valor = raiz->elemento;
    }
}

/* Función privada para agregar un elemento al árbol binario.
 */
template <typename Elem>
void Arbin<Elem>::agregarElemento(Nodo * arbol, Elem elemento)
{
    if (elemento <= arbol->elemento) {
        // Insertar en rama izquierda
        if (arbol->subIzquierdo != NULL) {
            agregarElemento(arbol->subIzquierdo, elemento);
        } else {
            Nodo * nuevo;
            nuevo = new Nodo;
            nuevo->elemento = elemento;
            nuevo->subIzquierdo = NULL;
            nuevo->subDerecho = NULL;

            arbol->subIzquierdo = nuevo;
        }
    } else {
        // Insertar en rama derecha
        if (arbol->subDerecho != NULL) {
            agregarElemento(arbol->subDerecho, elemento);
        } else {
            Nodo * nuevo;
            nuevo = new Nodo;
            nuevo->elemento = elemento;
            nuevo->subIzquierdo = NULL;
            nuevo->subDerecho = NULL;

            arbol->subDerecho = nuevo;
        }
    }
}

/* Función pública para agregar un elemento al árbol binario.
 */
template <typename Elem>
```

```
void Arbin<Elem>::agregarElemento(Elem elemento)
{
    if (raiz == NULL) {
        // El arbol está vacío
        Nodo * nuevo;
        nuevo = new Nodo;
        nuevo->elemento = elemento;
        nuevo->subIzquierdo = NULL;
        nuevo->subDerecho = NULL;

        raiz = nuevo;
    } else {
        agregarElemento(raiz, elemento);
    }
}

/* Función privada para listar los elementos del árbol inorden.
 */
template <typename Elem>
void Arbin<Elem>::listarInorden(Nodo * arbol, Lista<Elem> * I\& lista) const
{
    if (arbol != NULL) {
        listarInorden(arbol->subIzquierdo, lista);
        lista->agregarFinalLista(arbol->elemento);
        listarInorden(arbol->subDerecho, lista);
    }
}

/* Función pública para listar los elementos del árbol inorden.
 */
template <typename Elem>
void Arbin<Elem>::listarInorden(Lista<Elem> * I\& lista) const
{
    if (!vacioArbin()) {
        listarInorden(raiz, lista);
    }
}

/* Función privada para determinar si un elemento está en el árbol.
 */
template <typename Elem>
void Arbin<Elem>::estaIncluido(Nodo * arbol, Elem valor, bool I\& existe) const
{
    if (arbol != NULL) {
        if (arbol->elemento == valor) {
            existe = true;
        } else {
            if (valor < arbol->elemento) {
                estaIncluido(arbol->subIzquierdo, valor, existe);
            } else {
                estaIncluido(arbol->subDerecho, valor, existe);
            }
        }
    }
}
```

```
/* Función pública para determinar si un elemento está en el árbol.
 */
template <typename Elem>
bool Arbin<Elem>::estaIncluido(Elem valor) const
{
    if (vacioArbin()) {
        return false;
    } else {
        bool existe = false;

        estaIncluido(raiz, valor, existe);
        return existe;
    }
}

/* Función privada para contar la cantidad de elementos que tiene el árbol.
 */
template <typename Elem>
int Arbin<Elem>::cantidadNodos(Nodo * arbol) const
{
    if (arbol == NULL) {
        return 0;
    } else {
        return (1 + cantidadNodos(arbol->subIzquierdo) + cantidadNodos(arbol->subDerecho));
    }
}

/* Función pública para contar la cantidad de elementos que tiene el árbol.
 */
template <typename Elem>
int Arbin<Elem>::cantidadNodos() const
{
    return cantidadNodos(raiz);
}

/* Función privada para determinar la profundidad del árbol.
 */
template <typename Elem>
int Arbin<Elem>::profundidad(Nodo * arbol) const
{
    if (arbol == NULL) {
        return 0;
    } else {
        return (1 + max(profundidad(arbol->subIzquierdo), profundidad(arbol->subDerecho)));
    }
}

/* Función pública para determinar la profundidad del árbol.
 */
template <typename Elem>
int Arbin<Elem>::profundidad() const
{
    return profundidad(raiz);
}

/* Función para determinar si un nodo es hoja.
```

```
 */
template <typename Elem>
bool Arbin<Elem>::esHoja(Nodo * nodo) const
{
    return (nodo != NULL and nodo->subIzquierdo == NULL and nodo->subDerecho == NULL);
}

/* Función privada para construir la frontera del árbol.
 */
template <typename Elem>
void Arbin<Elem>::construirFrontera(Nodo * nodo, Lista<Elem> * I\& frontera) const
{
    if (nodo != NULL) {
        if (esHoja(nodo)) {
            frontera->agregarFinalLista(nodo->elemento);
        } else {
            construirFrontera(nodo->subIzquierdo, frontera);
            construirFrontera(nodo->subDerecho, frontera);
        }
    }
}

/* Función privada para construir la frontera del árbol.
 */
template <typename Elem>
void Arbin<Elem>::construirFrontera(Lista<Elem> * I\& frontera) const
{
    if (!vacioArbin()) {
        construirFrontera(raiz, frontera);
    }
}

// Tipos para los cuales la clase está implementada
template class Arbin<unsigned int>;
template class Arbin<int>;
template class Arbin<float>;
template class Arbin<char>;
template class Arbin<string>;
```