Estructuras de Datos y Algoritmos 1

TAD LISTA

Carlos Luna

Definición

Hemos visto la definición inductiva de listas.

Más precisamente, definimos inductivamente los tipos Alista, para cualquier tipo A (que es el tipo de los elementos de las listas en cuestión).

Los tipos Alista se definen inductivamente por medio de dos constructores:

[ ] : Alista

__ . __ : A * Alista -> Alista

Conceptos básicos

- En esta notación, indicamos los tipos de las operaciones: qué argumentos requieren y qué tipo de objetos producen.
- Se entiende entonces que, por definición, las Alistas son todos aquellos objetos que pueden ser formados combinando estas dos operaciones.

Ahora tratamos de presentar las listas como un TAD.

Genericidad

La primera observación es que la definición inductiva dada arriba no introduce un tipo, sino más bien un formador de tipos:

lista: Tipo -> Tipo

o, dicho de otra manera, todos los tipos Alista para cualquier tipo A.

En C++ usaremos templates.

TADs y Clases Abstractas

Vamos a introducir un tipo abstracto como una clase abstracta de C++
(declarada en un *header file*).

Al hacerlo, obtendremos un módulo que introduce al tipo abstracto de Listas de elementos de cualquier tipo.

El módulo de definición (o especificación) lo escribimos como sigue:

Operaciones

```
template <class Etype>
class Lista
{ public:
        < operaciones >
}
```

Qué operaciones tendremos?

Un conjunto mínimo queda dado por el siguiente criterio:

CONSTRUCTORAS: se corresponden con las operaciones introducidas en la definición inductiva.

Constructoras

Sin embargo las definiremos como métodos void de la clase abstracta List:

```
<operaciones>
virtual void Vacia()=0;
// construye la lista vacía

virtual void Cons(const Etype &x)=0;
// inserta x al comienzo de la lista

<…>
```

Predicados

En particular, para determinar si una lista es vacía o no, debemos aplicar a ella un método.

Este es un caso que se da con generalidad en tipos inductivos con más de un constructor:

- cada constructor corresponde a un caso o forma de elemento del tipo inductivo.
- para determinar si un elemento es de una cierta forma, usamos una función booleana.

Operación esVacía

- Estas son además casos de PREDICADOS sobre los elementos del tipo abstracto (corresponden con propiedades de los elementos).
- En el caso de las listas, hace falta un predicado para decidir entre las dos formas posibles de listas, vacía o no vacía:

```
< operaciones >
virtual bool esVacía()=0;
// retorna true si la lista es vacía
<...>
```

Operaciones selectoras

Finalmente, debemos poder acceder a la información contenida en las listas. Sólo las listas no vacías contienen información y ésta es, naturalmente, el valor de la cabeza y la cola de la lista.

Para descomponer listas no vacías usamos dos funciones.

Estos son dos casos de las usualmente llamadas (operaciones) SELECTORAS:

Cabeza y Cola

```
< operaciones >
virtual Etype Cabeza()=0;    // podria retornar Etype*
/* precondición: lista no vacía.
   retorna: la cabeza (primer elemento) */

virtual void Cola()=0;
/* precondición: lista no vacía,
   retorna: la cola de la lista */
<--->
```

Así concluye la especificación -mínima- de las Alistas como tipo abstracto:

```
// módulo ListaT.h
template <class Etype>
class Lista
{ public:
      virtual void Vacia()=0;
        // retorna la lista vacía
      virtual void Cons(const Etype &x)=0;
        // inserta x al comienzo de la lista
      virtual bool esVacia()=0;
        // retorna true si la lista es vacía
      virtual Etype Cabeza()=0;
        /* precondición: lista no vacía.
           retorna: la cabeza (primer elemento) */
      virtual void Cola()=0;
        /* precondición: s no vacía,
          retorna: la cola de la lista */
};
```

Especificación suficiente

El método de selección de las operaciones constructoras, predicados, selectoras puede aplicarse a todo tipo definido inductivamente.

El resultado es una especificación suficiente, en el sentido que toda otra operación sobre listas puede definirse en términos de las primitivas enumeradas arriba.

Ejemplo:

La función largo

```
template <class Etype>
int
largo (Lista<Etype> *l) {
  if (l->esVacia())
     return 0;
  else {  l->Cola();
          return 1 + largo(l); }
};
```

Observación

Es importante remarcar que no existe un único TAD Lista (una única especificaión de Lista).

Para Listas, como para muchos TADs, existe más de una especificación, con operaciones diferentes.

Por ejemplo, si consideramos la inserción de elementos en una lista podríamos incluir operaciones que insertan al comienzo de una lista, al final, luego de una posición corriente, ….

Implementación

- Como se ha dicho, una implementación de un TAD consiste en un tipo de estructura de datos concreto que se elige como representación del TAD y las correspondientes implementaciones de los procedimientos.
- En el caso de las listas especificadas arriba, la implementación natural se basa en la representación por medio de listas encadenadas.

Implementación en C++

- Una implementación de un TAD la entenderemos como la definición de una (sub)clase concreta de la clase abstracta en la que se definen las operaciones del TAD:

```
// módulo "ListaTimp.cpp"
<importaciones> //obligatorio #include "ListaT.h"
template <class Etype>
class ListaImpLEnc : public Lista<Etype>
{ <valores>
  <definición de operaciones(métodos)>
};
<implementación de operaciones>
```

Representación de objetos

- Comenzaremos por introducir el tipo mediante el cual representaremos los objetos del tipo abstracto:

```
<valores>
protected:
/* private: */
   struct NodoL {
       Etype elem;
       NodoL *sig;
   };
   typedef NodoL *PtrNodoL;
   PtrNodoL thelist, poscorr;
   void BorrarLista();
<--->
```

Representación de objetos (cont.)

- **`protected`**: al igual que en Java, acceso permitido a métodos de la clase y subclases.
- Un elemento de tipo **`NodoL`** esta formado por un elemento de tipo **`Etype`** y un puntero al siguiente nodo en la lista.
- **`thelist`** y **`poscorr`** son dos variables asociadas a un objeto de tipo Lista. La primera indica el comienzo de la lista encadenada y la segunda la “cabeza actual” del objeto (la *posición corriente*).

Métodos de `ListaImpLEnc`

```
<definición de operaciones>
 public:
      // Operaciones del TAD

      void Vacia();
      virtual void Cons(const Etype &x);
      // discutir el sentido de este virtual
      bool esVacia();
      Etype Cabeza();
      void Cola();

      // Constructor y Destructor C++
      ListaImpLEnc(){ Vacia(); }
      ~ListaImpLEnc { BorrarLista(); }

<--->
```

Constructores

```
<implementación de operaciones>

template <class Etype>
void
ListaImpLEnc<Etype> ::
Vacia() { thelist = poscorr = NULL;}

template <class Etype>
void
ListaImpLEnc<Etype> ::
Cons(const Etype &x) {
   PtrNodoL aux = new NodoL;
   if (aux == NULL) cout << "Out of Space";
   else{
       aux->elem = x; aux->sig = thelist;
       thelist = poscorr = aux; };
};
<...>
```

Constructores

- La operación **`Vacia()`** setea a **`thelist`** y **`poscorr`** en **`NULL`**.

- La operación **`Cons(x)`** inserta el elemento **`x`** al comienzo de la lista, quedando este nodo además como la posición corriente.

- Se podría definir también una operación **`Insertar(x)`**, cuyo efecto sería insertar a **`x`** luego de la posición corriente, dejando a este nuevo nodo como la posición corriente.

Predicado y Selectoras

```
<Implementación de operaciones>

template <class Etype>
bool
ListaImpLEnc<Etype> ::
esVacia() {return poscorr == NULL;}

template <class Etype>
Etype
ListaImpLEnc<Etype> ::
Cabeza() { return poscorr -> elem;}

template <class Etype>
void
ListaImpLEnc<Etype> ::
Cola() { poscorr = poscorr -> sig;}
<--->
```

Predicado y Selectoras (cont.)

- La implementación de `esVacia` es trivial. Notar que la operación considera a la posición corriente poscorr y no a thelist.
- Notar que `Cabeza()` retorna la información de la posición corriente.
- La implementación de `Cola` que modifica la posición corriente, definiéndola como el resto de la posición actual, nos permitirá acceder a todos los elementos del valor de un objeto Lista sin modificar el mismo.

Manejo de Precondiciones

Alternativas para manejar precondiciones en las operaciones:

- Al implemetar una operación con precondición, asumir que la misma se cumple (la responsabilidad es del usuario del TAD). Como se asumió en las transparencias anteriores.
- Considerar casos erroneos (por ejemplo, con un "if"). Esto lleva a totalizar una opreación parcial. El implementador realiza el manejo de los errores en el uso de las precondiciones.

Manejo de Precondiciones

- Uso de la macro assert de C++.

assert (*expresion*);

prueba el valor de una expresión. Si el valor de la expresión es 0 (falso), entonces assert imprime un mensaje de error (conteniendo el número de línea y el nombre del archivo), y llama a la función *abort* para terminar la ejecución del programa.

Esta es una herramienta de depuración útil.

Versión incorrecta de `BorrarLista`

```
template <class Etype>
void
ListImpLEnc<Etype>::
BorrarLista() {
   PtrNodoL aux = thelist;
   while (aux != NULL)
   { delete aux;
     aux = aux -> sig;
   };
   delete thelist;
};
```

Destruimos **`aux`** y luego intentamos usarlo!!

Versión correcta de **BorrarLista**

```
template <class Etype>
void
ListImpLEnc<Etype>::
BorrarLista() {
   PtrNodoL aux = thelist, temp;
   while (aux != NULL)
   { temp = aux -> sig;
     delete aux;
     aux = temp;
   };
   delete thelist;
   thelist = poscorr = NULL;
};
```

Ejemplo de uso del Tad `Lista<Etype>`

```
#include <iostream.h>
#include "ListaTimp.cpp"

void main
{
   Lista<int>* l = new ListImpLenc<int>;
   for (int i = 1, i<= 5, i++)
       l->Cons(i);

   cout << "La lista es: 5 4 3 2 1";
   while (! (l->esVacia()))
   {   cout << l->Cabeza() << "\n";
       l->Cola();
   };
}
```

Ejercicios propuestos

- Extender el TAD Lista con las operaciones:
  - **`ListaVacia()`**: similar a la operación **`esVacia()`** pero sobre la lista (no sobre la posición corriente).
  - **`insertar(x)`**: referida previamente.
  - **`comienzo()`**: modifica la posición corriente de la lista, sitúandola al comienzo.
  - **`borrar_prim()`**: elimina el primer elemento de la lista (corresponde a la operación **`Cola()`** que modifica la lista). Deja la posición corriente al comienzo.
  - **`borrar_corr()`**: elimina el elemento de la lista que se encuentra en la posición corriente. Deja la posición corriente en el próximo.

Notar que las dos últimas operaciones tienen precondiciones.

TAD´s acotados y no acotados

Los TAD´s pueden ser acotados o no en la cantidad de elementos.

En general, las implementaciones estáticas refieren a una versión del TAD que es acotada (incluso en la especificación del TAD),

mientras que las implementaciones dinámicas admiten una versión del TAD no acotada (que permite expresar la noción de estructuras arbitrariamente grandes).

Lo cierto es que TAD´s acotados y no acotados son conceptualmente diferentes.

TAD´s acotados y no acotados

Los TAD´s acotados incorporan una operación en su especificación que permite testar si la estructura está llena.

virtual bool isFull()

Asismimo, se agrega una precondición a la especificación de la operación de inserción del TAD (“que la estructura no esté llena”) .

A nivel de C++, la especificación acotada de un TAD es conceptualmente una subclase de la no acotada y puede implementarse así (es por esto que las operaciones que agregan elementos suelen definirse como virtual en las implementaciones).

Implementaciones de Listas

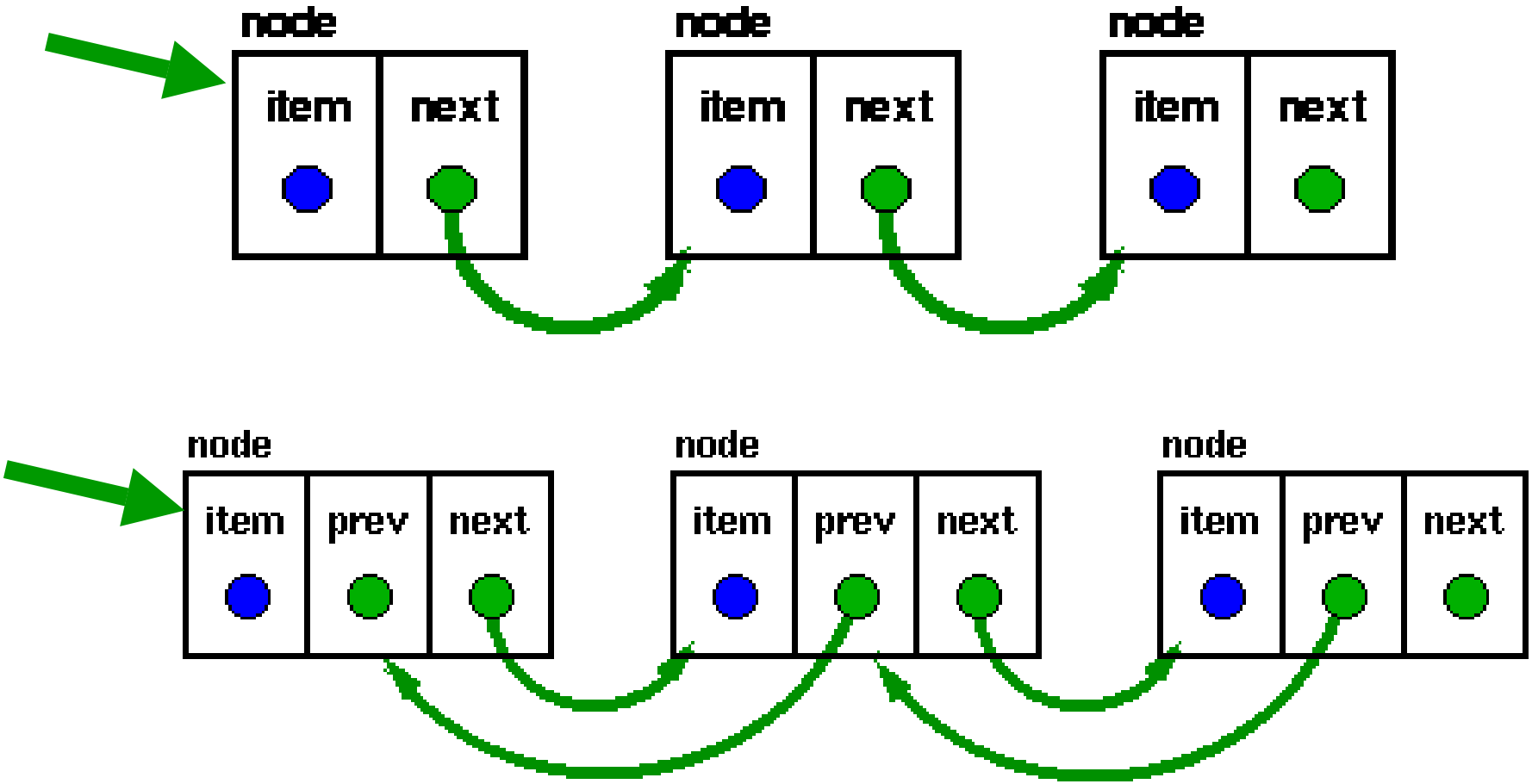
NOTA:

Vimos una implementación dinámica de Listas.

En el práctico se trabajará sobre ésta y otras implementaciones de listas,

- dinámicas, como por ejemplo listas simple y doblemente encadenadas, y
- estáticas, con el uso de arreglos.

Implementaciones dinámicas de Listas



Bibliografía

– Data Structures and Algorithm Analysis in C++

Mark Allen Weiss; Benjamin/Cummings Inc., 1994.

– Estructuras de Datos y Algoritmos

A. Aho, J. E. Hopcroft & J. D. Ullman; Addison-Wesley, 1983.

– Como Programar en C/C++ (o versión sólo C++)

H.M. Deitel & P.J. Deitel; Prentice Hall, 1994.

Anexo 1: Una especificación de listas

/**

Especificación de Lista (EspecLista.h).

Aclaraciones:

Este archivo de especificación es para listas no acotadas, donde las implementaciones dinámicas son las más adecuadas. (es una posible especificación)

En caso de que se desee trabajar con listas acotadas (cuyas implementaciones son habitualmente basadas en estructuras de datos estáticas), se puede crear un nuevo archivo que incluya a éste (y herede de la clase Lista) agregándole el predicado: virtual bool isFull() y adicionando a las operaciones de inserción, como precondición, que la lista no este llena.

Anexo 1: Una especificación de listas

El "=0" al final del perfil de cada función indica que la misma es una función virtual pura creando así una clase abstracta.

Las directivas al preprocesador:

```
#ifndef LISTA_H
#define LISTA_H
.....
codigo
.....
#endif
```

previenen de error al tratar de incluir el archivo cuando ha sido previamente incluido en otro archivo.

Anexo 1: Una especificación de listas

Sugerencia: Cada vez que se desee emplear esta técnica se puede especificar el nombre del archivo en mayúsculas separando su extensión con el subguión.

El método Cons retorna bool para controlar si se pudo insertar el nuevo elemento.

```
**********************************************************/
#ifndef LISTA_H
#define LISTA_H
template <class T>
class Lista{
public:
    //CONSTRUCTORAS
    virtual void Vacia()=0;  //Construye la lista vacia
```

Anexo 1: Una especificación de listas

```
virtual bool Cons(const T &x)=0;
/* Inserta el elemento x luego de la posición corriente
en la lista. Retorna true si el elemento fue insertado
correctamente o false si el elemento no pudo
agregarse. La posicion corriente es el elemento
recien insertado */

//SELECTORAS
virtual T Elemento()=0;
/* PreCondicion: La posición corriente no está al final
de la lista (en particular, la lista es no vacia).
Retorna el elemento en la posicion corriente */
```

Anexo 1: Una especificación de listas

virtual void Inicio()=0;

//Coloca la posicion corriente al inicio de la lista

virtual void Siguiente()=0;

/* PreCondicion: Lista no vacia.

Mueve la posicion corriente al siguiente nodo, en caso de que la posicion corriente sea el final de la lista, coloca la posicion corriente al inicio de la lista (tiene un comportamiento circular). */

//PREDICADOS

virtual bool esVacia()=0;

//Retona true si la lista esta vacia, sino, retorna false

Anexo 1: Una especificación de listas

```
    virtual bool esFinal()=0;
    //Retorna true si la posicion actual es el final de la lista

    //OTRAS
    virtual void Borra()=0;
    /* PreCondicion: La posición corriente no está al final
    de la lista (en particular, la lista es no vacia).
    Borra el elemento de la posicion corriente. La posicion
    corriente es la siguiente al nodo eliminado. */

    virtual void BorrarLista()=0;
    /* Vacia la lista. */
};
 #endif
```

Anexo 2: Una Implementación dinámica de listas

```
// ImpListaEnlazada.cpp
#ifndef IMPLISTAENLAZADA_CPP
#define IMPLISTAENLAZADA_CPP
#include "EspecLista.h”
#include <assert.h>
template <class T>  class ListaEnlazada : public Lista<T>
{  //Definicion de Valores
    private:
        struct NodoL { T dato; NodoL *sig; };
        typedef NodoL *PtrNodoL;
        PtrNodoL theList;
        PtrNodoL pos;
```

Anexo 2: Una Implementación dinámica de listas

```
// Definicion de operaciones
public: //Constructor y destructor
        ListaEnlazada() { Vacia(); };
        virtual ~ListaEnlazada () {
                BorrarLista();
                delete [] pos;
                delete [] theList; };
//Operaciones del TAD
//CONSTRUCTORAS
void Vacia() {
        theList = new NodoL();
        theList -> sig = NULL;
        pos = theList; };
```

Anexo 2: Una Implementación dinámica de listas

```
virtual bool Cons(const T &x) {
    PtrNodoL aux = new NodoL();
    if (aux == NULL) return 0;
            // No hay memoria suficiente para crear el nodo.
    else {  if (esVacia()) {
                    aux -> dato = x; aux -> sig = NULL;
                    theList -> sig = aux; pos = aux; }
            else {
                    aux -> dato = x; aux -> sig = pos -> sig;
                    pos -> sig = aux; pos = pos -> sig; }
            return 1; // El nodo se creo correctamente. }
};
```

Anexo 2: Una Implementación dinámica de listas

```
//SELECTORAS
T Elemento() { assert(pos!=NULL); return pos -> dato; };


void Inicio() { pos = theList -> sig; };

void Siguiente() {
      if(!esVacia()) { if(!esFinal()) pos = pos -> sig;
                       else  pos = theList; }
};

//PREDICADOS
bool esVacia() { return (theList -> sig == NULL); };
bool esFinal() { return (pos == NULL); };
```

Anexo 2: Una Implementación dinámica de listas

```
//OTRAS
void Borra() {
    assert(!(esVacia()));
    PtrNodoL temp = pos;
    Inicio();
    if(pos == temp) { //El nodo a borrar esta al inicio
            theList -> sig = pos -> sig;
            Siguiente();
            delete [] temp; }
    else { //El nodo a borrar esta en otro lugar de la lista
            while ((pos -> sig != temp) && (!(esFinal())))
                    Siguiente();
            pos -> sig = temp -> sig; delete [] temp; }
};
```

Anexo 2: Una Implementación dinámica de listas

```
    void BorrarLista() {
        while (theList != NULL) {
            pos = theList;
            theList = pos -> sig;
            delete [] pos; }
        Vacia();
    };

};
#endif
```

Anexo 3: Uso de Listas (Pruebas)

```
/*************************************************************
    Archivo de prueba de Lista  Aclaraciones:
    Si se usan los simbolos de < y > en los includes, el archivo
    se va a buscar a los directorios donde C almacena las
    librerias por defecto.  En cambio, si se usan las " " el
    archivo se va a buscar en el directorio donde este el
    proyecto.
    *************************************************************/
    #include <iostream.h>
    #include "ImpListaEnlazada.cpp”
    Lista<int>* l = new ListaEnlazada<int>();
    void VerificoVacia() {
          if (l -> esVacia()) cout << "La lista esta Vacia" << "\n";
          else cout << "La lista no esta Vacia" << "\n"; };
```

Anexo 3: Uso de Listas (Pruebas)

```
void PruebaInsercion() {
    VerificoVacia();
    for (int i = 1; i <= 10; i++) {
            cout << "Inserto elemento: " << i << "\n"; l ->Cons(i); }
            VerificoVacia(); };

void PruebaRecorrido() {
    l -> Inicio();
    if (!(l -> esVacia())) {
            while (!(l ->esFinal())) {
                    cout << "Muestro elemento: " << l ->Elemento() << "\n";
                    l ->Siguiente(); }
    } else { cout << "No se puede recorrer la lista" << "\n";
              VerificoVacia(); }
};
```

Anexo 3: Uso de Listas (Pruebas)

```
void PruebaBorrado1()
    { VerificoVacia();
    // Pruebo con casos borde
    //Elementos al inicio de la lista

    l -> Inicio(); cout << "Borro el elemento: " << l ->Elemento() << "\n";
    l -> Borra();

    //Elementos al final de la lista
    while(l ->Elemento() != 10) l -> Siguiente();
    cout << "Borro el elemento: " << l ->Elemento() << "\n";
    l -> Borra();
```

Anexo 3: Uso de Listas (Pruebas)

```
    //Elementos "medios" de la lista
    l ->Inicio(); while(l ->Elemento() != 4)  l -> Siguiente();
    cout << "Borro el elemento: " << l ->Elemento() << "\n"; l -> Borra();

    while(l ->Elemento() != 7) l -> Siguiente();
    cout << "Borro el elemento: " << l ->Elemento() << "\n"; l -> Borra();};

    void PruebaBorrado2() { VerificoVacia();
    cout << "Borro la lista completa" << "\n";
    l -> BorrarLista(); VerificoVacia();
};
    void main() {
          PruebaInsercion(); PruebaRecorrido(); PruebaBorrado1();
          PruebaRecorrido(); PruebaBorrado2(); PruebaInsercion();
          PruebaRecorrido(); };
```