13. LISTAS CON PUNTO DE INTERÉS

13.1. Introducción

Las listas son secuencias de elementos a_1, a_2, a_3,... ,a_{n-1}, a_n. Diremos que el tamaño de la secuencia es 'n'. Si las listas tienen cero elementos diremos que son listas vacías.

En una lista diremos que el elementos a_{i+1} sigue o sucede al elemento a_i (si $i < n$), y diremos que el elemento a_{i-1} precede o es anterior a a_i (si $i > 1$).

Operaciones habitualmente ligadas a listas son: Crear la lista vacía; añadir elementos a la lista (en cualquier posición); eliminar elementos (de cualquier posición); moverse por la lista;... Aunque se pueden especificar otras según las necesidades de la aplicación, como buscar un determinado elemento o mostrar todos los elementos de la lista.

Si nos damos cuenta las operaciones que se definen nos relacionan claramente las listas con las pilas y las colas, siendo las listas generalizaciones de estas (las operaciones de inserción y borrado pueden realizarse en cualquier punto de la secuencia y no sólo sobre los extremos de la misma.)

13.2. Listas con punto de interés

Existen diferentes modelos de uso e implementación de las listas y uno de los más interesantes y extendidos son las llamadas listas con punto de interés.

En las listas con punto de interés se define la existencia de un determinado elemento 'destacado' que es el que servirá de referencia para la realización de todas las operaciones. Este elemento destacado puede modificarse con la aplicación de otras funciones, y vendrá marcado como punto de interés de la lista (y lo señalaremos con un *.) Puede suceder que el punto de interés esté además de sobre cualquier elemento de la lista, a continuación del último, fuera de la secuencia. Esta situación del punto de interés servirá para realizar operaciones al final de la lista, y diremos en este caso que el punto de interés está *totalmente a la derecha* de la lista. A ese punto le llamaremos desde este momento "Final de la lista" (y lo dibujaremos como ⊕.) Sin embargo no se podrá situar el punto de interés por delante del primer elemento de la lista, a lo sumo sobre él.

Las operaciones que vamos a definir sobre las listas con punto de interés, serán similares a las vistas ya con pilas y colas: Crear la estructura vacía, añadir información a la estructura, eliminar información de la estructura, consultar información de la estructura y consultar la existencia de información en la estructura. Pero como en este caso las operaciones no se realizan sólo en uno de los extremos de la estructura o en puntos concretos de la estructura, habrá que añadir operaciones

especificas para elegir el punto de la estructura en donde realizar las operaciones (operaciones para mover el punto de interés de la lista.)

En adelante, y en todo el tema, siempre que hablemos de Listas estaremos refiriéndonos a listas con punto de interés.

Estructura

Lista (Valor) {* Valor será el tipo de datos que podremos guardar en la lista *}

Operaciones

/* Operaciones similares a las vistas en Pilas y Colas */
INICIAR_LISTA ( ) → Lista
INSERTAR (Lista , Valor) → Lista
ELIMINAR (Lista) → Lista
CONSULTA (Lista) → Valor
LISTA_VACIA (Lista) → Lógico
/* Operaciones específicas */
IR_A_INICIO (Lista) → Lista
AVANZAR (Lista) → Lista
FINAL_LISTA (Lista) → Lógico

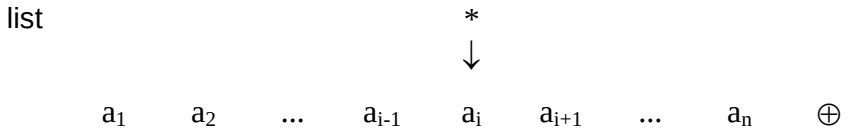
Especificación informal

INICIAR_LISTA ( ) → Lista

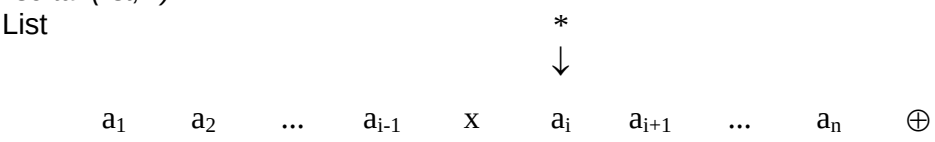
Inicia la lista a vacía, con solo el punto 'final de la lista' y el punto de interés sobre este 'final de la lista'

INSERTAR (Lista , Valor) → Lista

Inserta el Valor delante de elemento señalado como punto de interés, dejando el punto de interés sobre el elemento en el que estaba.

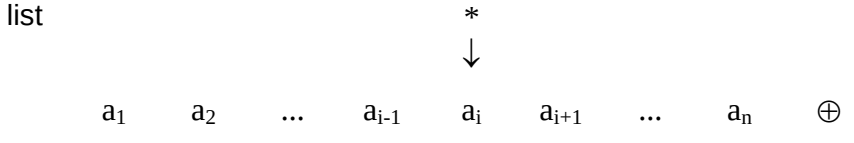


Insertar (list, x)

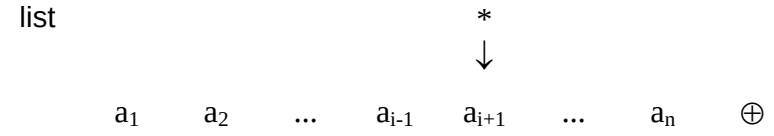


ELIMINAR (Lista) → Lista

Modifica la lista eliminando el elemento señalado por el punto de interés, dejando el punto de interés sobre el elemento siguiente al eliminado. Si el punto de interés señalaba el final de la lista devolverá error.

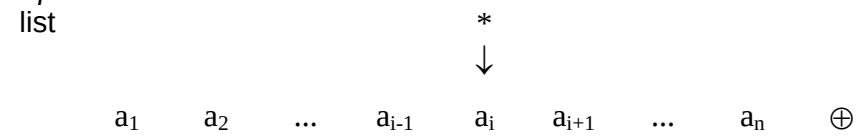


Eliminar (list)



CONSULTAR (Lista) → Valor

Devuelve la información señalada por el punto de interés. Si el punto de interés señalaba el final de la lista devolverá error.

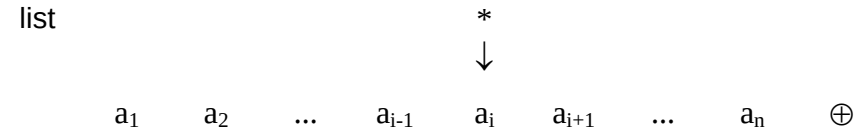


Consultar (list) → a_i

LISTA_VACIA (Lista) → Lógico

Nos informa si existe información en la lista o no (nos dice si el número de elementos de la lista es cero o diferente de cero.)

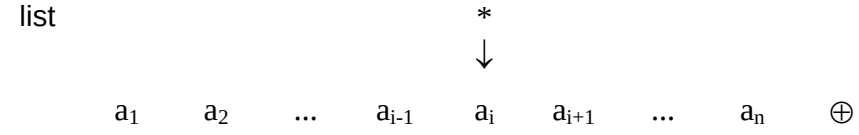
Lista_Vacia (list) → true



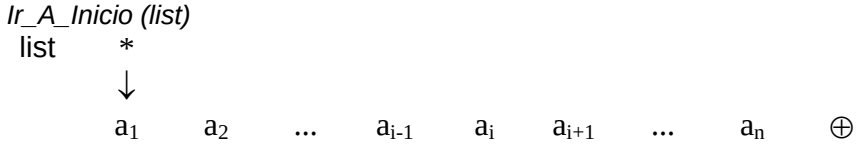
Lista_Vacia (list) → false

IR_A_INICIO (Lista) → Lista

Mueve el punto de interés al primer elemento de la lista.

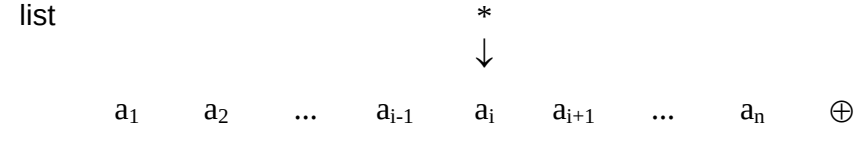


Ir_A_Inicio (list)

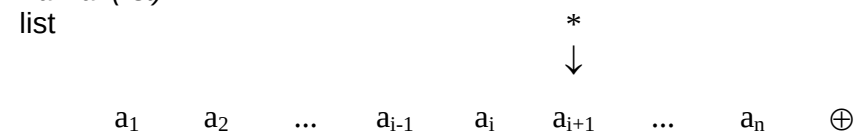


AVANZAR (Lista) → Lista

Avanza el punto de interés al siguiente elemento de la lista.



Avanzar (list)



FINAL_LISTA (Lista) → Lógico

Nos dice si el punto de interés está o no al final de la lista.

List *

↓

a_1 a_2 ... a_{i-1} a_i a_{i+1} ... a_n ⊕

Final_Lista (list) → false

List *

↓

a_1 a_2 ... a_{i-1} a_i a_{i+1} ... a_n ⊕

Final_Lista (list) → true

Algunos axiomas

∀ list ∈ Lista, x ∈ Valor se cumple que:

LISTA_VACIA (INICIAR_LISTA ( )) → cierto

LISTA_VACIA (INSERTAR (list, x)) → falso

ELIMINAR (INICIAR_LISTA ( )) → error

Si CONSULTAR (list) = x entonces

INSERTAR (ELIMINAR (list), x) → list

FINAL_LISTA (INICIAR_LISTA ( )) → cierto

FINAL_LISTA (INSERTAR (list , x) → | cierto Si (FINAL_LISTA (list) = cierto) | falso Si (FINAL_LISTA (list) = falso)

AVANZAR (INICIAR_LISTA ( )) → error

...

Una vez vistas las operaciones, su comportamiento en la descripción informal y algunos de los axiomas que las rigen, ya podemos plantearnos la utilización del tipo abstracto de datos Lista.

Ejemplo de utilización de listas:

1. Realizar una función que busque un determinado valor en una lista.

```
Funcion Buscar (Lista list, Valor y): Boolean
Inicio
    Booleano encontrado;

    encontrado ← falso
    list ← Ir_A_Inicio (list)
    Mientras ( no Final_Lista (list) And no encontrado) Hacer
        Si ( Consultar (list) = y ) Entonces
            encontrado ← cierto
        Sino
            list ← Avanzar (list)
        Fin_si
    Fin_mientras

    Devolver (encontrado)
Fin
```

2. Realizar una función que obtenga la media de los elementos contenidos en una lista, suponiendo el tipo Valor como Real.

```
Funcion Media (Lista list): Real
Inicio
    Real acc, med;
```

```
    Entero num;

    acc ← 0
    num ← 0

    list ← Ir_A_Inicio (list)
    Mientras ( no Final_Lista (list) ) Hacer
        acc ← acc + Consultar (list)
        num ← num + 1
        list ← Avanzar (list)
    Fin_mientras

    Si (num = 0)
        med ← 0
    Sino
        med ← acc / num
    Fin_si

    Devolver (med)
Fin
```

Ejercicio:

3. Realizar un programa en pseudocódigo que pida valores enteros al usuario y los inserte ordenadamente en una lista de enteros (tras insertar, por ejemplo 4, 3, 6, 5, 1 la lista contendrá los valores en el orden correcto, es decir, 1, 3, 4, 5, 6.) El proceso se detendrá cuando introduzcamos un valor negativo.

13.3. Implementación de la clase LISTA en C++

Al igual que hicimos en los temas de Pilas y Colas, la implementación en C++ la haremos utilizando clases.

Recordar que la iniciación de la estructura a vacía, se la dejaremos al constructor de la clase, y que, al igual que hicimos en Pilas y Colas, las funciones que puedan producir errores devolverán booleanos que indicarán si la operación se ha producido satisfactoriamente o no.

```
class Lista
{
    public:
        typedef .??. Valor;

        Lista (void);
        Lista (const Lista &);
        ~Lista (void);
        bool Insertar (Valor);
        bool Eliminar (void);
        bool Consulta (Valor &);
        bool ListaVacia (void);

        void IrAInicio (void);
        bool Avanzar (void);
        bool FinalLista (void);

    private:
        ????
};
```

La especificación de la parte privada la dejaremos para el momento en que especifiquemos qué implementación utilizaremos de la Lista: Una implementación estática (utilización de vectores para

guardar la información) o una dinámica (mediante la utilización de punteros para crear una estructura enlazada).

Con esta especificación ya estamos en condiciones de realizar programas que hagan uso de esta parte pública y trabajen con Listas.

Ejemplo de utilización de listas con el interfaz propuesto en C++:

1. Realizar una función que busque un determinado valor en una lista.

```
bool Buscar (Lista list, Lista::Valor y)
{
      bool encontrado;
      Lista::Valor x;

      encontrado = false;

      list.IrAInicio ( );
      while ( !list.FinalLista ( ) && !encontrado )
      {
            list.Consultar (x);
            if (x == y)
                  encontrado = true;
            else
                  list.Avanzar ( );
      }

      return encontrado;
}
```

2. Realizar una función que obtenga la media de los elementos contenidos en una lista, suponiendo el tipo Valor como Real.

```
float Media (Lista list)
{
    Lista::Valor x;
    float acc, med;
    int num;

    acc = 0;
    num = 0;

    list.IrAInicio ( );
    while (!FinalLista (list) )
    {
        list.Consultar (x)
        acc = acc + x;
        num = num + 1;
        list.Avanzar ( )
    }

    if (num == 0)
        med = 0;
    else
        med = acc / num;

    return med;
}
```

13.3.1 Implementación de la clase LISTA en C++ con vectores

En primer lugar implementaremos las listas utilizando como tipo base un vector. Para guardar la información contenida en la lista, utilizaremos el vector, pero también necesitaremos guardar la información de cuántos elementos contiene la lista. La manera en que guardaremos la información será compactando los elementos en las primeras posiciones del vector, de manera que la lista siempre comenzará en la posición cero.

Con estas consideraciones, y sabiendo que necesitamos una marca para localizar el elemento destacado, la parte privada de la clase quedaría como sigue:

```
class Lista
{
    public:
        ...

    private:
        typedef Valor Vector[MAX];

        Vector info;
        int fin;
        int pto;
};
```

`MAX` es una constante que marca la capacidad máxima de la lista y **`Valor`** el tipo de información que puede contener la lista.

Con esta definición de la parte privada de la lista los métodos se implementarán de la siguiente manera.

Iniciación: Constructor de la clase

Funcion INICIAR_LISTA
Salida
Lista
Inicio
fin ← 0
pto ← 0
Fin

Esta operación, en C++, se convierte en el constructor por defecto:

```
Lista::Lista (void)
{
   fin = 0;
   pto = 0;
}
```

Operación de inserción de información

La operación de inserción tendrá que abrir un hueco en la posición del punto de interés e insertar adecuadamente la información. Al finalizar la operación tendremos que el punto de interés apuntará al siguiente elemento al que apuntaba (pues hemos insetado en la posición en la que apuntaba) y además tendremos un elemento más en la lista.

Además la función devolverá error si ya no caben más elementos en la lista.

```
Funcion INSERTAR ( Lista , Valor ) → Lista
Entradas
   list: Lista
   x: Valor
Salida
   list
Inicio
   Si (fin = MAX) entonces
      Escribir ("Error: Lista llena")
   Sino
      Desde i ← fin hasta pto + 1 hacer
         info[i] ← info[i – 1]
      Fin_desde
      info[pto] ← x
      pto = pto + 1
      fin = fin + 1
   Fin_si
Fin
```

Codificada en C++:

```
bool Lista::Insertar (x)
{
   bool error;
   int i;

   if (fin == MAX)
      error = true;
   else
   {
      error = false;

      for (i = fin; i > pto; i--)
         info[i] = info[i – 1];

      info[pto] = x;
      pto++;
      fin++;
   }
   return error;
}
```

Operación de eliminación de información

Se eliminará la información apuntada por el punto de interés. En caso en que el punto de inetrés apunte a un punto de la lista que no contiene información válida (el punto de interés apunta a fin) devolveremos error.

```
Funcion ELIMINAR
Entrada
   list: Lista
Salida
   list
Inicio
   Si (pto = fin) entonces
        Escribir ("Error: Imposible eliminar elemento")
   Sino
        Desde i ← pto hasta fin - 2 hacer
             info[i] ← info[i + 1]
        Fin_desde
        fin  ← fin – 1
   Fin_si
Fin
```

```
bool Lista::Eliminar (void)
{
   bool error;
   int i;

   if (pto == fin)
      error = true;
   else
   {
      error = false;

      for (i = pto; i < fin – 1; i++)
         info[i] = info[i + 1];
      fin--;
   }
   return error;
}
```

Operación de consulta de información

Hay que devolver el valor apuntado por el punto de interés. Al igual que en la elimnación tendremos un error si el punto de interés apunta a un punto que no contenga información.

```
Funcion CONSULTAR
Entrada
   list: Lista
Salida
   x: Valor
Inicio
   Si (pto = fin) entonces
        Escribir ("Error: Imposible eliminar elemento")
   Sino
```

```
            x ← info[pto]
        Fin_si
    Fin
```

```
bool Lista::Consultar (Valor & x)
{
   bool error;

   if (pto == fin)
      error = true;
   else
   {
      error = false;

      x = info[pto];
   }
   return error;
}
```

Operación de consulta sobre la propia lista: Lista Vacia

```
    Funcion LISTA_VACIA
    Entrada
        list: Lista
    Salida
        Lógico
    Inicio
        Si (fin = 0) entonces
            b_aux ← TRUE
        Sino
            b_aux ← FALSE
        Sino

        Devolver (b_aux)
    Fin
```

```
bool Lista::ListaVacia (void)
{
   bool b_aux;

   if (fin == 0)
      b_aux = true;
   else
      b_aux = false;

   return b_aux;
}
```

O lo que es lo mismo, podemos devolver el valor resultante de la comparación entre fin y cero

```
bool Lista::ListaVacia (void)
{
   return fin == 0;
}
```

Operaciones de modificación del punto de interés: IrAInicio

Mueve el punto de interés a la primera posición de la lista, en nuestro caso la posción cero del vector.

```
Funcion IR_A_INICIO
Entrada
   list: Lista
Salida
   list
Inicio
   pto ← 0
Fin
```

```
bool Lista::IrAInicio (void)
{
   pto = 0;
}
```

Operaciones de modificación del punto de interés: Avanzar

Mueve el punto de interés a la siguiente posición válida de la lista. Si está al final de la lista devolverá error.

```
Funcion AVANZAR
Entrada
   list: Lista
Salida
   list
Inicio
   Si (pto = fin)
      Escribir ("Error: Imposible avanzar punto de interés")
   Sino
      pto ← pto + 1
   Fin_si
Fin
```

```
bool Lista::Avazar (void)
{
   bool error;

   if (pto == fin)
      error = true;
   else
   {
      error = false;

      pto = pto + 1;
   }
   return error;
}
```

Operaciones de consulta del punto de interés: Final Lista

Funcion FINAL_LISTA
Entrada
list: Lista
Salida
Lógico
Inicio
Si (pto = fin) **entonces**
b_aux ← **VERDADERO**
Sino
b_aux ← **FALSO**
Fin_si
Devolver (b_aux)
Fin

```
bool Lista::FinalLista (void)
{
   bool b_aux;

   if (pto == fin)
      b_aux = true;
   else
      b_aux = false;

   return b_aux;
}
```

O como en el caso de la consulta de la propia lista, podemos devolver el resultado de la comparación entre el punto de interés y fin.

```
bool Lista::FinalLista (void)
{
   return pto == fin;
}
```

13.3.2 Implementación de la clase LISTA en C++ con cursores

Vista la implementación de la clase Lista con vectores en C++, se puede ver que en programas en los que exista una gran cantidad de inserciones y eliminaciones serán muy lentos debido al coste lineal de estas operaciones. Por ello se puede estudiar la posibilidad de no tener que mover todos los elementos cada vez que se realicen operaciones de este tipo.

Una posibilidad sencilla es la utilización de cursores.

Un cursor va a ser una referencia a un índice de un vector, de manera que la relación de sucesión entre dos elementos de la lista vendrá dada por la sucesión de índices del vector. Por ejemplo, supongamos que tenemos la lista de elementos a_1, a_2, a_3, a_4, a_5 y a_6 dispersos en un vector como muestra la figura:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
a_4	a_1		a_2	a_5			a_3	a_6	

La sucesión correcta de cursores sería: 1, 3, 7, 0, 4 y 8, mientras que el resto de posiciones del vector no se utilizan. Una manera de implementar esta situación es añadir un campo en el vector que indique cuál es el siguiente elemento en la sucesión.

El primer elemento estará marcado con un índice, al que llamaremos inicio, y marcaremos el final de la secuencia indicando que el siguiente elemento es el de índice '-1'. El punto de interés será, al igual que en el caso de vectores, simplemente un índice del vector.

Gráficamente la estructura quedaría como sigue:

		0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
ini	*1*		a_4	a_1		a_2	a_5			a_3	a_6	
pto	*¿?*		**4**	**3**		**7**	**8**			**0**	**-1**	

donde '*ini*' es el índice donde empieza la secuencia, y '*pto*' el índice que indica el punto de interés (que en este caso podría coger los valores 0, 1, 3, 4, 7, 8 o -1).

Visto esto, la parte privada de la clase quedaría como sigue:

```
class Lista
{
   public:
      ...

   private:
      struct Nodo
      {
         Valor info;
         int cursor;
      };

      typedef Nodo Vector[MAX];

      Vector info;
      int ini;
      int pto;
};
```

A partir de esto es fácil implementar los métodos propios de la clase. Sólo resaltar un par de detalles: Sería interesante poder discriminar fácilmente las casillas ocupadas por información válida de las que están 'vacías' para hacer más simple la inserción. esto lo podríamos hacer poniendo en el campo del cursor un valor 'no válido', por ejemplo MAX o '-2'. También sería interesante para la

inserción detrás del último (cuando el punto de interés está situado detrás del último) tener un índice que señalase al último elemento.

A partir de estas recomendaciones, se plantea como ejercicio la realización de los métodos (recordar que, al igual que en el caso de la implementación con vectores, tanto el constructor de copia como el destructor de clase no son necesarios, y no habría que ponerlos en la interfaz de la clase).

13.3.3 Implementación de la clase LISTA en C++ con punteros

Realmente, la implementación con cursores, es una primera aproximación a la implementación con punteros. La única diferencia es que mientras que con cursores la 'memoria' ocupada por la estructura está limitada al tamaño del vector y el control de 'memoria' ocupada y libre está en manos del programador, con punteros, la memoria está gestionada por el sistema, de manera que no tenemos que localizar los huecos ni marcarlos como libres, sino que es el sistema, a través de **`new`** y **`delete`** se encarga de esta gestión.

En principio tendríamos que una lista queda determinada por el puntero que marca su inicio y el puntero que marca el elemento destacado o punto de interés. Veremos más adelante que es interesante para realizar algunas de las operaciones, marcar el último de los elementos. De manera que necesitaremos tres punteros para determinar exactamente la Lista:

```
class Lista
{
   public:
      ...

   private:
      struct Nodo;
      typedef Nodo* Puntero;
      struct Nodo
      {
         Valor info;
         Puntero sig;
      };

      Puntero ini;   /* Puntero al inicio de la lista */
      Puntero fin;   /* Puntero al inicio de la lista */
      Puntero pto;   /* Punto de interés              */
};
```

Con está representación los métodos de la clase Lista quedarían como sigue:

Iniciación: Constructor de la clase

Los objetos de la clase quedan iniciados poniendo el inicio de la lista a Null y el punto de interés en ese elemento.

```
Lista::Lista ()
{
    ini = NULL;
    fin = NULL;   // Realmente no es necesaria esta iniciación
    pto = NULL;
}
```

El constructor de copia

Copia en la lista del objeto 'actual' la lista pasada por referencia. Será similar al constructor de copia realizado con colas, pero teniendo en cuenta que también hay que copiar el punto de interés.

```
Lista::Lista (const Lista & orig)
{
    Puntero aux, dup;

    ini = NULL;
    fin = NULL;
    pto = NULL;

    aux = orig.inicio;

    while (aux != NULL)
    {
        dup = new Nodo;
        dup->info = aux->info;
        dup->sig = NULL;
        if (inicio == NULL)    //si vacia
            ini = dup;
        else                   //detras del ultimo
            fin->sig = dup;
        fin = dup;          //siempre es el ultimo

        if (aux == orig.pto)   //fijar pto. de interes
            pto = dup;
        aux = aux->sig;
    }
}
```

Destructor de la clase

Tiene que liberar todos los elemento de la lista dinámica.

```
Lista::~Lista (void)
{
    Puntero p_aux;

    while (ini != NULL)
    {
        p_aux = ini;
        ini = ini->sig;
        delete p_aux;
    }
    fin = NULL;
    pto = NULL;
}
```

Inserción delante del punto de interés

En principio, para insertar delante del punto de interés hay que conocer el elemento anterior al que señale el punto de interés. Primero haremos una búsqueda de ese elemento (si existe) y a partir de este elemento insertaremos. En este caso supondremos que siempre se puede reservar memoria para el nuevo elemento.

```
bool Lista::Insertar (Valor x)
{
   Puntero p_aux;
   bool error;

   error = false;

   p_aux = new Nodo;

   p_aux->Info = x;
   p_aux->sig = pto;

   /* Si la inserción es delante de la cabeza */
   if (pto == ini)
   {
      ini = p_aux;
   }
   /* En cualquier otro caso */
   else
   {
      p_aux2 = ini;
      while (p_aux2->sig != pto)
         p_aux2 = p_aux2->sig;

      p_aux2->sig = p_aux;
   }
}
```

Esta inserción, en general, es lineal. Se puede mejorar haciendo una pequeña 'trampa' que consiste básicamente en insertar el nuevo elemento detrás del punto de interés y cambiar a este nuevo elemento la información del punto de interés y modificar la información contenida en el punto de interés con la información que hay que insertar:

```
bool Lista::Insertar (Valor x)
{
    bool exito;
    Puntero aux;

    exito = true;
    aux = new Nodo;
    if (pto == NULL)        //insertar al final
    {
        aux->info = x;
        aux->sig = NULL;
        if (ini == NULL)    //si vacia
            ini = aux;
        else                //detras del ultimo
            fin->sig = aux;
        fin = aux;          //siempre es el ultimo
    }
    else
    {
        *aux = *pto;
        pto->info = x;
        pto->sig = aux;
        if (pto == fin)  //caso particular
            fin = aux;
        pto = aux;
    }
    return exito;
}
```

Eliminación del punto de interés

Al igual que en la inserción, para eliminar el punto de interés hay que localizar el elemento anterior al punto de interés. Una vez hecho esto, modificar los enlaces, modificar el punto de interés y 'desreservar' el espacio ocupado

```
bool Eliminar (void)
{
   Puntero p_aux, ant;
   bool error;

   if (pto == NULL)
      error = false;
   else
   {
      error = true;

      /* Si queremos borrar la cabeza */
      if (pto == ini)
      {
         ini = pto->Sig;
         delete pto;
         pto = ini;
      }
      /* En cualquier otro caso */
      else
      {
         ant = NULL;
         p_aux = ini;
         while (p_aux->sig != pto)
         {
            ant = p_aux;
            p_aux = p_aux->sig;
         }

         if (pto == fin)
            fin = ant;
         p_aux->sig = pto->sig;
         delete pto;
         pto = p_aux;
      }
   }
   return error;
}
```

Como sucedía en la inserción, en general, esta eliminación es lineal. Se puede mejorar haciendo una pequeña 'trampa' que consiste básicamente en eliminar el elemento que está tras el punto de interés.

```
bool Lista::Eliminar ()
{
    Puntero aux;
    bool exito;

    if ( (pto == NULL) || ListaVacia() )
        exito = false;
    else
    {
        exito = true;
        if (pto == fin)     //no hay nada detrás
        {
```

```
            if (pto == ini) //es el único
            {
                delete pto;
                ini = NULL;
                fin = NULL;
                pto = NULL;
            }
            else
            {
                aux = ini;
                /*
                 * buscar el penúltimo
                 */
                while (aux->sig != fin)
                    aux = aux->sig;
                fin = aux;
                fin->sig = NULL;
                delete pto;
                pto = NULL;
            }
        }
        else
        {
            /*
             * caso normal
             */
            aux = pto->sig;
            *pto = *aux;
            if (aux == fin)
                fin = pto;
            delete aux;
        }
    }
    return exito;
}
```

Consulta

Se limitará a devolver el valor contenido en el elemento destacado, si éste no es el 'final de la lista'.

```
bool Lista::Consulta (Valor & x)
{
   bool exito;

   if (pto == NULL)
      exito = false;
   else
   {
      exito = true;

      x = pto->Info;
   }

   return error;
}
```

Lista Vacía

Comprueba si en la lista existe algún elemento además del 'final de la lista'.

```
bool Lista::ListaVacia (void)
{
   return ini == NULL;
}
```

Mover al inicio el punto de interés

Pone el punto de interés en el primer elemento de la lista.

```
void Lista::IrAInicio (void)
{
   pto = ini;
}
```

Avanzar el punto de interés

Avanza el punto de interés al siguiente elemento de la lista si el punto de interés no estaba ya en el punto más a la derecha de la lista.

```
bool Lista::Avanzar (void)
{
   bool exito;

   if (pto == NULL)
      exito = false;
   else
   {
      exito = true;

      pto = pto->sig;
   }
   return error;
}
```

Comprobar si el punto de interés está al final de la lista

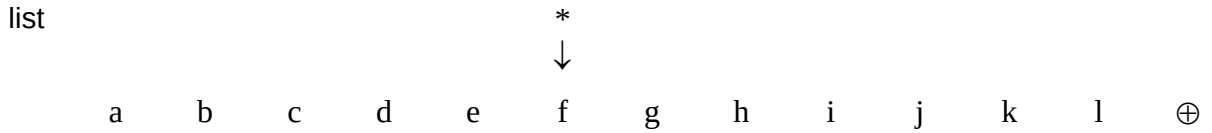
Nos dice si el punto de interés está sobre el punto más a la derecha de la lista (sobre el final de la lista.)

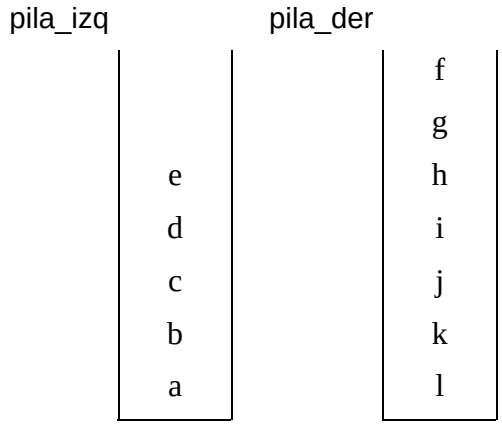
```
bool Lista::FinalLista (void)
{
  return pto == NULL;
}
```

13.3.4 Implementación de la clase LISTA en C++ con pilas

Otra posible implementación de las listas con punto de interés es mediante la utilización de otro tipo abstracto de datos: Pilas.

Una manera de ver las listas con punto de interés es como dos pilas enfrentadas, de manera que el punto de interés está situado sobre la cima de la pila de la derecha.





Veamos, con esta filosofía como se implementarían cada una de las operaciones públicas de la clase:

```
Lista (void);
Lista (const Lista &);
~Lista (void);

bool Insertar (Valor x);
bool Eliminar (void);
bool Consulta (Valor & x);
bool ListaVacia (void);

void IrAInicio (void);
bool Avanzar (void);
bool FinalLista (void);
```

Respecto a los constructores y el destructor de la clase:

Constructor por defecto: Como la clase contiene en su parte privada dos pilas, el hecho de declarar una lista supone declarar las dos pilas, que tendrán asociado su propio constructor por defecto.

Constructor de copia: Al pasar una lista por valor, ésta contendrá una copia de las dos pilas, que llamarán a sus propios constructores de copia si es necesario.

Destructor de la clase: Al acabar la vida del objeto 'Lista', acabará la vida de las pilas, que llamarán a su propio destructor si es necesario.

En conclusión, tanto los constructores de la clase, como el destructor no necesitan realizar ninguna tarea.

Respecto de la información de la lista:

Insertar un nuevo elemento supondrá apilar en la pila de la izquierda.

Eliminar, se limita a desapilar de la pila de la derecha.

Consultar el punto de interés solo es mirar la cima de la pila de la derecha.

Y diremos que la lista está vacía si lo están ambas pilas.

Respecto del punto de interés:

IrAInicio es poner toda la información que hay en la pila de la izquierda, sobre la pila de la derecha.

Avanzar el punto de interés supondrá sólo desapilar de la pila de la derecha y apilar a la izquierda.

Y diremos que el punto de interés está al final de la lista si la lista de la derecha está vacía.

Con estas consideraciones es fácil implementar las listas con punto de interés basándonos en pilas.

Variantes de las implementaciones de listas enlazadas

En la implementación dinámica de las listas hemos visto que existían diferentes casos especiales que complicaban ligeramente la implementación de los métodos. Para evitar estos casos especiales existen diferentes mejoras que pueden aplicarse por separado o combinadas en la implementació según nos interese mejorar ciertos aspectos de la implementación.

Listas doblemente enlazadas

Hemos visto que en algunos de los algoritmos teníamos el problema de que no se conocia el elemento anterior al punto de inserción. Este hecho nos llevaba a tener que hacer métodos que en general eran lineales, o modificar el método haciendo la inserción real detrás del elemento, pero intercambiando la información de cada elemento.

Esta situación se podría resolver fácilmente si conociesemos el elemento anterior de uno dado.

La manera de saberlo es mediante la utilización de un puntero adicional en cada nodo que proporcione esa información.

Con esta modificación ciertas tareas mejorarían (aumentaría la eficiencia temporal) pero el tamaño de cada elemento aumenta (disminuye la eficiencia espacial de la estructura.)

Listas circulares

Otra posible mejora en ciertos casos es la conversión de la lista lineal, en circular, es decir que el siguiente elemento, tras el último vuelva a ser el primero. Con esto se consigue que no exista el puntero a NULL al final de la lista y evita el caso particular en que el punto de interés apunta a un elemento no válido.

Listas con nodo cabeza

Finalmente, otro de los problemas que hemos tenido en la implementación de las listas enlazadas (dinámicas) ha sido el tener que comprobar en cada momento si la lista tenía o no elementos.

Si nos aseguramos que la lista siempre tinen un elemento, esta comprobación sería innecesaria.

La solución sería insertar al inicio de la lista un elemento ‘ficticio’ que no contiene realmente ninguna información válida, pero que evitaría que la lista vacía fuese sólo un puntero apuntando a NULL.

A continuación se muestra la implementación en C++ de una estructura que combinaría todas estas mejoras: una lista doblemente enlazada, circular y con nodo cabeza.

La interfaz de la clase sería:

```
class Lista
{
    public:
        Lista (void);
        Lista (const Lista &);
        ~Lista (void);
        bool Insertar (Valor);
        bool Eliminar (void);
        bool Consulta (Valor &);
        bool ListaVacia (void);

        void IrAInicio (void);
        bool Avanzar (void);
        bool FinalLista (void);

    private:
        struct Nodo;
        typedef Nodo* Puntero;
```

```
        struct Nodo
        {
            Valor info;
            Puntero sig;
            Puntero ant;
        };

        Puntero cab;   /* Puntero al inicio de la lista */
        Puntero pto;   /* Punto de interés              */
};
```

Y la implementación de los diferentes métodos:

```
/*
 * Constructor por defecto
 */
Lista::Lista (void)
{
    cab = new Nodo;
    cab->sig = cab;
    cab->ant = cab;
}

/*
 * Constructor de copia
 */
Lista::Lista (const Lista & ori)
{
    Puntero p_aux, q_aux;

    /* Se crea la lista vacía */
    cab = new Nodo;
    cab->sig = cab;
    cab->ant = cab;

    /* Se recorre la lista original */
    p_aux = ori.cab->sig;
    while (p_aux != ori.cab)
    {
        /* Se reserva hueco y actualiza la informacion */
        q_aux = new Nodo;
        q_aux->info = p_aux->info;

        /*
         * Y se van añadiendo los elementos delante de la cabeza
         * es decir al final de la lista
         */
        q_aux->sig = cab;
        q_aux->ant = cab->ant;
        cab->ant->sig = q_aux;
        cab->ant = q_aux;

        /*
         * Si estamos copiando el pto de interés lo fijamos en
         * la copia
         */
        if (p_aux == ori.pto)
            pto = q_aux;

        p_aux = p_aux->sig;
    }
}
```

```
/*
 * Destructor de la clase
 */
Lista::~Lista (void)
{
    Puntero p_aux;

    while (cab != cab->sig)
    {
        p_aux = cab->sig;
        cab->sig = p_aux->sig;
        delete p_aux;
    }
    delete cab;
}

/*
 * Inserción de nuevos elementos delante del punto de interés
 */
bool Lista::Insertar (Valor x)
{
    bool error = false;

    q_aux = new Nodo;
    q_aux->info = x;

    q_aux->sig = pto;
    q_aux->ant = pto->ant;

    pto->ant->sig = q_aux;
    pto->ant = q_aux;

    return error;
}

/*
 * Eliminación del elemento apuntado por el punto de interés
 */
bool Lista::Eliminar (void)
{
    bool error;

    if (pto == cab)
        error = true;
    else
    {
        error = false;

        p_aux = pto->sig;

        pto->sig->ant = pto->ant;
        pto->ant->sig = pto->sig;

        delete pto;

        pto = p_aux;
    }
    return error;
}
```

```
/*
 * Consulta del elemento apuntado por el punto de interés
 */
bool Lista::Consulta (Valor & x)
{
    bool error;

    if (pto == cab)
        error = true;
    else
    {
        error = false;
        x = pto->info;
    }

    return error;
}

/*
 * Comprobación de si existen elementos en la lista además de la cabeza
 */
bool Lista::ListaVacia (void)
{
    return cab == cab->sig;
}

/*
 * Situar el punto de interés en el primer elemento válido de la lista
 */
void Lista::IrAInicio (void)
{
    pto = cab->sig;
}

/*
 * Avanzar el punto de interés si no hemos llegado al final de la lista
 */
bool Lista::Avanzar (void)
{
    bool error;

    if (pto == cab)
        error = true;
    else
    {
        error = false;
        pto = pto -> sig;
    }
}

/*
 * Comprobar que hemos llegado al final de la lista
 * (comprobar que de nuevo estamos en la cabeza)
 */
bool Lista::FinalLista (void)
{
    return pto == cab;
}
```