Departamento
de Ingeniería de la
Información y las
Comunicaciones
UMU

Tecnología de la Programación

Título de Grado en Ingeniería Informática

Curso 2023/24

Grupo 4

Tema 2. Estructuras de Datos Enlazadas Lineales

Dept. Ingeniería de la Información y las Comunicaciones

Universidad de Murcia

Tema 2. Estructuras de Datos Enlazadas Lineales

2.1 Motivación del uso de estructuras de datos enlazadas
2.2 Estructuras de datos enlazadas lineales
2.3 Ejercicios resueltos
2.4 Ejercicios propuestos

2.1 Motivación del uso de estructuras de datos enlazadas

Multitud de aplicaciones requieren **estructuras de datos cuyo tamaño puede ir cambiando a lo largo de la aplicación**.

En estos casos, las representaciones contiguas (por ejemplo, con arrays) no son eficientes ya que el tamaño de éstas permanece invariable y aunque existen operaciones que permiten la modificación del tamaño (por ejemplo, `realloc`), el tiempo de ejecución de estas operaciones puede ser directamente proporcional al número de datos almacenados.

Se requieren entonces estructuras de datos que puedan crecer o decrecer según las necesidades de la aplicación, para lo cual se utilizan ***estructuras de datos enlazadas***. Una estructura de datos enlazada permite, con un **enunciado finito**, representar un **número potencialmente infinito de datos** enlazados entre sí.

Por otro lado, las **inserciones y eliminaciones** en estructuras contiguas pueden suponer reubicaciones de elementos para gestionar huecos, lo que requiere un tiempo adicional. Las estructuras de datos enlazadas permiten insertar y eliminar elementos en la estructura en un tiempo constante, ya que sólo hay que **cambiar los enlaces correspondientes**.

2.2 Estructuras de datos enlazadas lineales (1/34)

El lenguaje de programación *C* utiliza **apuntadores junto con tipos incompletos** para declarar estructuras de datos enlazadas. La siguiente declaración define el tipo `NodoPtr` para representar una ***estructura de datos enlazada lineal con simple enlace*** en donde cada elemento, de un tipo homogéneo `Elemento`, se enlaza con un elemento siguiente mediante un apuntador:

```
//versión 1
typedef int Elemento;
struct Nodo
{
      Elemento elem;
      struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;
```

```
//versión 2
typedef int Elemento;
typedef struct Nodo * NodoPtr;
struct Nodo
{
      Elemento elem;
      NodoPtr sig;
};
```

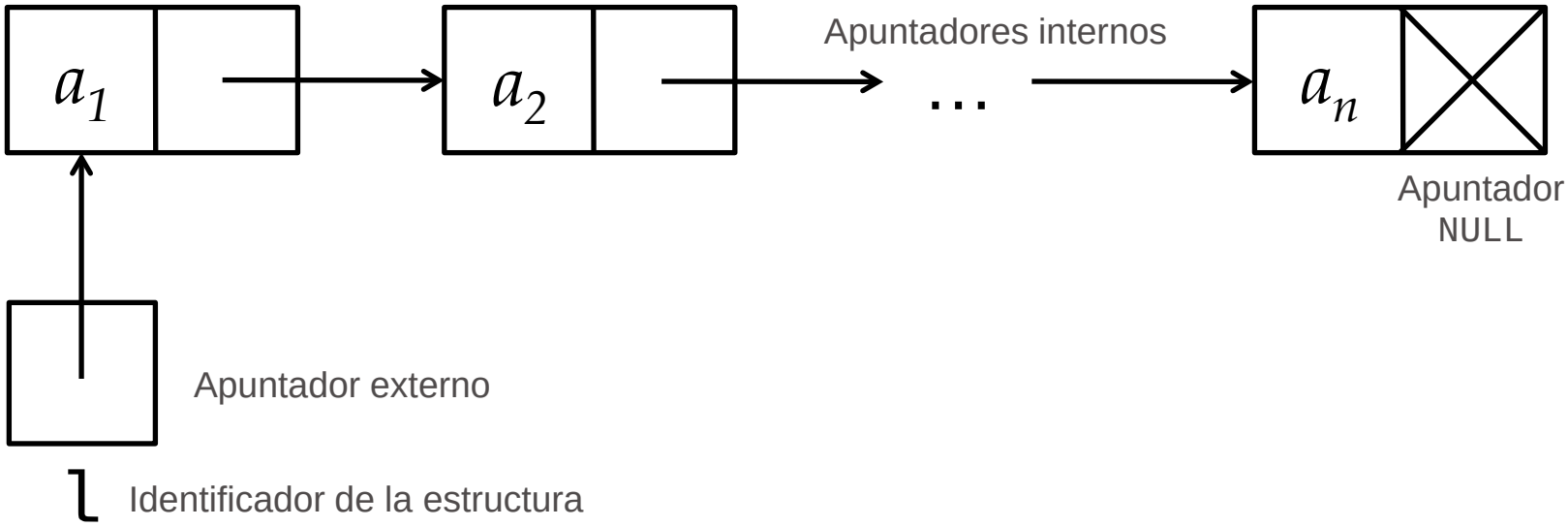
En las declaraciones anteriores, el tipo `NodoPtr` se ha definido como un apuntador a un tipo `struct Nodo`. El tipo `struct Nodo` está definido con el miembro `elem` para albergar un dato de tipo `Elemento` y con el miembro `sig` que es un apuntador al siguiente nodo. El tipo del miembro `sig` es por tanto un apuntador al tipo `struct Nodo`, que no está aún definido, por lo que es un apuntador a un tipo incompleto (en ambas versiones). En la versión 1, el tipo `NodoPtr` se define después del tipo `struct Nodo`, mientras que en la versión 2 se hace al revés, por lo que en la versión 2 el miembro `sig` puede definirse directamente de tipo `NodoPtr`.

Para crear una estructura enlazada, se declarará una variable del tipo `NodoPtr` que en tiempo de ejecución apuntará al primer nodo de la estructura enlazada. Esta variable será por tanto la conexión externa (puntero externo) con la estructura de datos enlazada. Un apuntador con valor `NULL` en el miembro siguiente de un nodo indica que dicho nodo es el último de la estructura de datos.

2.2 Estructuras de datos enlazadas lineales (2/34)

Representación gráfica de una estructura enlazada lineal con simple enlace

$$l = < a_1, a_2, \ldots, a_n >$$



2.2 Estructuras de datos enlazadas lineales (3/34)

Declaración e inicialización

```
NodoPtr l;
l = NULL;
```

Inserción por el principio

```
NodoPtr nuevo = malloc(sizeof(struct Nodo));
nuevo->elem = dato;
nuevo->sig = l;
l = nuevo;
```

NOTA: El código que aparece en esta diapositiva y en algunas de las siguientes está marcado en color rojo para indicar que el código es sólo explicativo para entender que la ausencia de un *nodo de encabezamiento* en estructuras enlazadas lineales produce un código menos legible y menos eficiente que mediante el uso de un *nodo de encabezamiento*. Dicho código marcado en rojo será finalmente sustituido por el equivalente mediante el uso de *nodo de encabezamiento*.

2.2 Estructuras de datos enlazadas lineales (4/34)

Inserción por el final

```
NodoPtr nuevo = malloc(sizeof(struct Nodo));
nuevo->elem = dato;
if (l == NULL) // Caso especial lista vacía
{
     nuevo->sig = NULL;
     l = nuevo;
}
else
{
     NodoPtr p = l;
     while (p->sig != NULL)
     {
          p = p->sig;
     }
     nuevo->sig = NULL;
     p->sig = nuevo;
}
```

2.2 Estructuras de datos enlazadas lineales (5/34)

Inserción en una posición predeterminada por un índice entero `i` (`1<= i <= longitud(l)+1`)

```
NodoPtr nuevo = malloc(sizeof(struct Nodo));
nuevo->elem = dato;
if (i == 1) // Caso especial inserción por el principio
{
    nuevo->sig = l;
    l = nuevo;
} else {
    NodoPtr p = l;
    int j = 1;
    while (j < i-1) {
        p = p->sig;
        j++;
    }
    nuevo->sig = p->sig;
    p->sig = nuevo;
}
```

2.2 Estructuras de datos enlazadas lineales (6/34)

Inserción a continuación de un nodo predeterminado p (de tipo `NodoPtr`) que apunta a un nodo de la estructura enlazada, o a `NULL` indicando inserción por el principio

```
NodoPtr nuevo = malloc(sizeof(struct Nodo));
nuevo->elem = dato;
if (p == NULL) // Caso especial inserción por el principio
{
     nuevo->sig = l;
     l = nuevo;
}
else
{
     nuevo->sig = p->sig;
     p->sig = nuevo;
}
```

2.2 Estructuras de datos enlazadas lineales (7/34)

Inserción antes de un nodo predeterminado `p`

INEFICIENTE porque requiere localizar la posición del nodo anterior a `p`, lo que implica un tiempo de ejecución de *O(N)* en el peor caso.

2.2 Estructuras de datos enlazadas lineales (8/34)

Inserción buscando la posición de inserción (por ejemplo, inserción de `dato` en orden creciente, asumiendo que la estructura está ordenada crecientemente)

```
NodoPtr nuevo = malloc(sizeof(struct Nodo));
nuevo->elem = dato;
if ((l == NULL) || (l->elem > dato))
{
     nuevo->sig = l;
     l = nuevo;
}
else
{
     NodoPtr p = l;
     while ((p->sig != NULL) && (p->sig->elem < dato))
     {
          p = p->sig;
     }
     nuevo->sig = p->sig;
     p->sig = nuevo;
}
```

2.2 Estructuras de datos enlazadas lineales (9/34)

Abstracción operacional (ejemplo de inserción por el principio)

Dado que el valor de `l` cambia tras la inserción, **se requiere un paso de parámetro por referencia**

```
void inserta(NodoPtr * l, Elemento dato)
{
     NodoPtr nuevo = malloc(sizeof(struct Nodo));
     nuevo->elem = dato;
     nuevo->sig = *l;
     *l = nuevo;
}
int main()
{
     int dato = 3;
     NodoPtr l = NULL;
     inserta(&l,dato);
     return 0;
}
```

2.2 Estructuras de datos enlazadas lineales (10/34)

Conclusiones:

- Tratamiento especial cuando la lista está vacía o la posición de inserción es la primera.
- La abstracción operacional requiere un paso de parámetros por referencia.

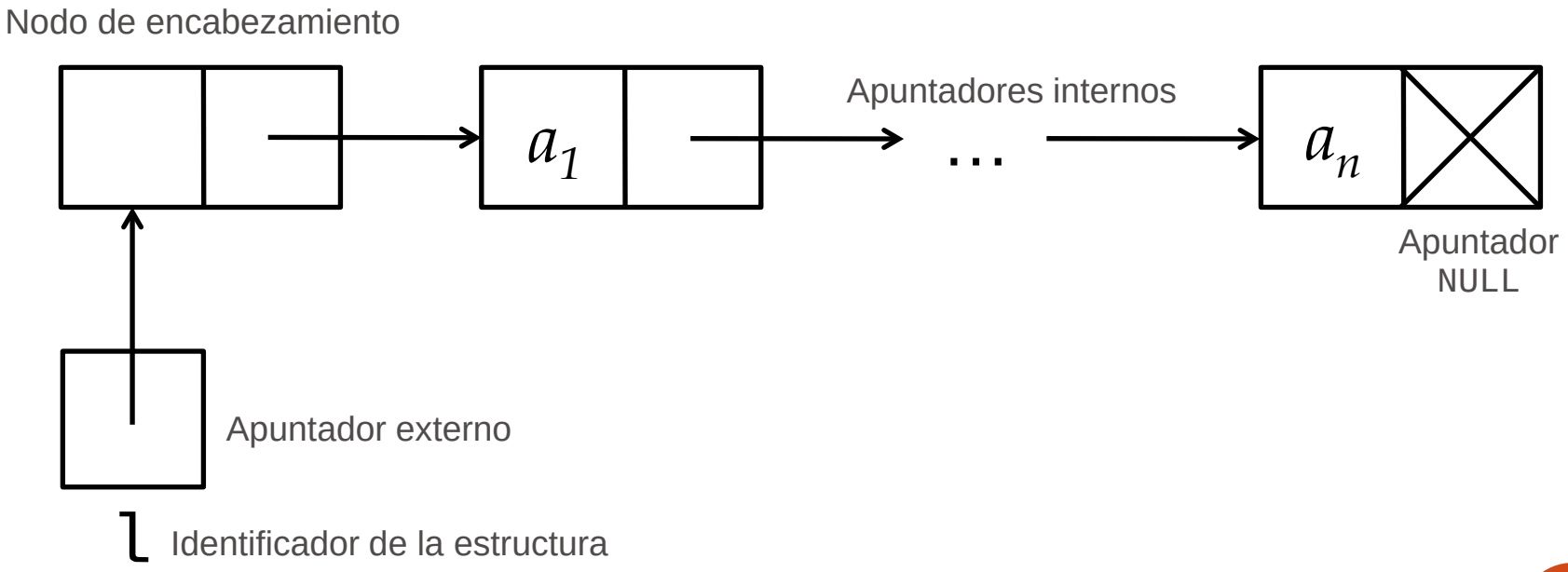
Alternativa para reducir la complejidad del código:

- Uso de uno ***nodo de encabezamiento*** al inicio de la estructura (también denominado ***cabecera*** o ***encabezado***).

2.2 Estructuras de datos enlazadas lineales (11/34)

Representación gráfica de una estructura enlazada lineal con simple enlace con nodo de encabezamiento

$$l = < a_1, a_2, \ldots, a_n >$$



2.2 Estructuras de datos enlazadas lineales (12/34)

Declaración y función de creación

```
NodoPtr crea()
{
     NodoPtr l = malloc(sizeof(struct Nodo));
     l->sig = NULL;
     return l;
}
int main()
{
     NodoPtr l = crea();
     // ...
     return 0;
}
```

2.2 Estructuras de datos enlazadas lineales (13/34)

Función de inserción por el principio

Dado que el valor de `l` apunta a la celda de encabezamiento desde su inicialización, no es necesario el paso por referencia.

```
void inserta(NodoPtr l, Elemento dato)
{
     NodoPtr aux = malloc(sizeof(struct Nodo));
     aux->elem = dato;
     aux->sig = l->sig;
     l->sig = aux;
}
int main()
{
     int dato = 3;
     NodoPtr l = crea();
     inserta(l,dato);
     // ...
     return 0;
}
```

2.2 Estructuras de datos enlazadas lineales (14/34)

Función de inserción por el final

```
void inserta_final(NodoPtr l, Elemento dato)
{
    NodoPtr p = l;
    while (p->sig != NULL)
        p = p->sig;
    p->sig = malloc(sizeof(struct Nodo));
    p->sig->elem = dato;
    p->sig->sig = NULL;
}
```

2.2 Estructuras de datos enlazadas lineales (15/34)

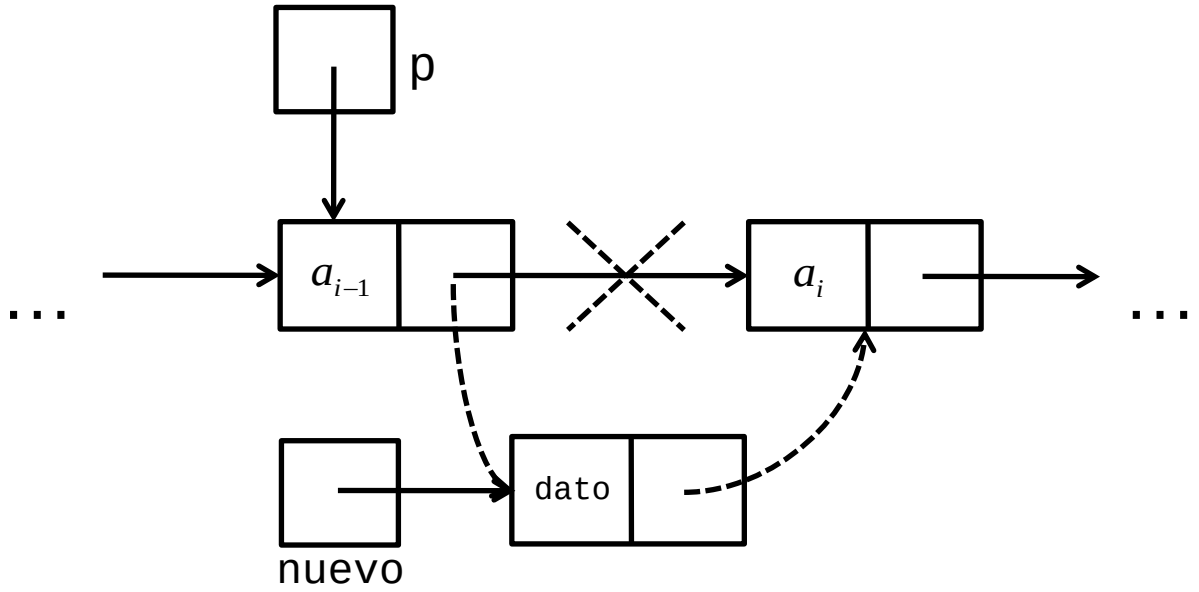
Función de inserción en una posición predeterminada por un índice entero

```
//req.: 1<=i<=longitud(l)+1
void inserta_indice(NodoPtr l, Elemento dato, int i)
{
     NodoPtr nuevo = malloc(sizeof(struct Nodo));
     nuevo->elem = dato;
     NodoPtr p = l;
     int j = 1;
     while (j < i)
     {
          p = p->sig;
          j++;
     }
     nuevo->sig = p->sig;
     p->sig = nuevo;
}
```

2.2 Estructuras de datos enlazadas lineales (16/34)

Función de inserción a continuación de un nodo predeterminado

```
void inserta_posicion(NodoPtr l, Elemento dato, NodoPtr p)
{
     NodoPtr nuevo = malloc(sizeof(struct Nodo));
     nuevo->elem = dato;
     nuevo->sig = p->sig;
     p->sig = nuevo;
}
```



2.2 Estructuras de datos enlazadas lineales (17/34)

Función de inserción buscando la posición de inserción

```
// req.: la estructura l está ordenada crecientemente
void inserta_ordenado(NodoPtr l, Elemento dato)
{
    NodoPtr p = l;
    while ((p->sig != NULL) && (p->sig->elem < dato))
        p = p->sig;
    NodoPtr nuevo = malloc(sizeof(struct Nodo));
    nuevo->elem = dato;
    nuevo->sig = p->sig;
    p->sig = nuevo;
}
```

2.2 Estructuras de datos enlazadas lineales (18/34)

Función de supresión del primer elemento

```
// req.: la estructura l contiene al menos un elemento
void suprime_primero(NodoPtr l)
{
    NodoPtr eliminado = l->sig;
    l->sig = eliminado->sig;
    free(eliminado);
}
```

2.2 Estructuras de datos enlazadas lineales (19/34)

Función de supresión del último elemento

```
// req.: la estructura l contiene al menos un elemento
void suprime_ultimo(NodoPtr l)
{
    NodoPtr p = l;
    while (p->sig->sig != NULL)
        p = p->sig;
    free(p->sig);
    p->sig = NULL;
}
```

2.2 Estructuras de datos enlazadas lineales (20/34)

Función de supresión de la primera ocurrencia de un elemento predeterminado

```
void suprime_primer_dato(NodoPtr l, Elemento dato)
{
    NodoPtr p = l;
    while ((p->sig != NULL) && (p->sig->elem != dato))
        p = p->sig;
    // Se suprime sólo si se ha encontrado
    if (p->sig != NULL)
    {
        NodoPtr eliminado = p->sig;
        p->sig = eliminado->sig;
        free(eliminado);
    }
}
```

2.2 Estructuras de datos enlazadas lineales (21/34)

Función de supresión de la primera ocurrencia de un elemento predeterminado en una estructura ordenada crecientemente

```
// req.: la lista l está ordenada crecientemente
void suprime_dato_ordenada(NodoPtr l, Elemento dato)
{
     NodoPtr p = l;
     while ((p->sig != NULL) && (p->sig->elem < dato))
          p = p->sig;
     // Se suprime sólo si se ha encontrado
     if ((p->sig != NULL) && (p->sig->elem == dato))
     {
          NodoPtr eliminado = p->sig;
          p->sig = eliminado->sig;
          free(eliminado);
     }
}
```

2.2 Estructuras de datos enlazadas lineales (22/34)

Función de supresión de todas las ocurrencias de un elemento predeterminado

```
void suprime_todos(NodoPtr l, Elemento dato)
{
     NodoPtr p = l;
     while (p->sig != NULL)
     {
          if (p->sig->elem == dato)
          {
               NodoPtr eliminado = p->sig;
               p->sig = eliminado->sig;
               free(eliminado);
          } else {
               p = p->sig;
          }
     }
}
```

2.2 Estructuras de datos enlazadas lineales (23/34)

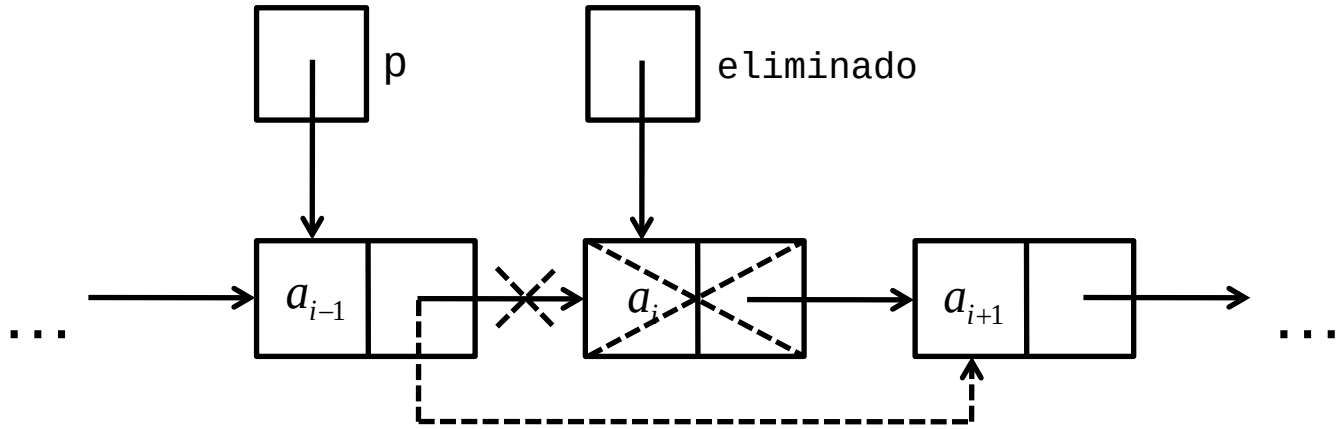
Función de supresión en una posición predeterminada por un índice entero

```
//req.: 1<=i<=longitud(l)
void suprime_indice(NodoPtr l, int i)
{
    NodoPtr p = l;
    int j = 1;
    while (j < i)
    {
        p = p->sig;
        j++;
    }
    NodoPtr eliminado = p->sig;
    p->sig = eliminado->sig;
    free(eliminado);
}
```

2.2 Estructuras de datos enlazadas lineales (24/34)

Función de supresión a continuación de un nodo predeterminado

```
// req.: la lista l contiene al menos un elemento
void suprime_posicion(NodoPtr l, NodoPtr p)
{
    NodoPtr eliminado = p->sig;
    p->sig = eliminado->sig;
    free(eliminado);
}
```



2.2 Estructuras de datos enlazadas lineales (25/34)

Función de impresión en pantalla

```
void imprime(NodoPtr l)
{
    l = l->sig;
    while (l != NULL)
    {
        printf("%d\n", l->elem);
        l = l->sig;
    }
}
```

2.2 Estructuras de datos enlazadas lineales (26/34)

Función de búsqueda de un elemento determinado

```
int busqueda(NodoPtr l, Elemento dato)
{
    l = l->sig;
    while ((l != NULL) && (l->elem != dato))
        l = l->sig;
    return l != NULL;
}
```

2.2 Estructuras de datos enlazadas lineales (27/34)

Función de búsqueda del índice de un elemento determinado

```
int busqueda_indice(NodoPtr l, Elemento dato)
{
    l = l->sig;
    int i = 1;
    while ((l != NULL) && (l->elem != dato))
    {
        l = l->sig;
        i++;
    }
    if (l != NULL) return i;
    return -1;
}
```

2.2 Estructuras de datos enlazadas lineales (28/34)

Función de búsqueda de la posición anterior a la de un elemento determinado

```
NodoPtr busqueda_posicion_ant(NodoPtr l, Elemento dato)
{
    while ((l->sig != NULL) && (l->sig->elem != dato))
        l = l->sig;
    if (l->sig != NULL) return l;
    return NULL;
}
```

2.2 Estructuras de datos enlazadas lineales (29/34)

Función de impresión en pantalla en orden inverso

```
// O(N²)
void imprime_inverso(NodoPtr l)
{
    NodoPtr ultimo = NULL;
    while (l->sig != ultimo)
    {
        NodoPtr p = l;
        while (p->sig != ultimo)
            p = p->sig;
        printf("%d\n", p->elem);
        ultimo = p;
    }
}
```

2.2 Estructuras de datos enlazadas lineales (30/34)

Función de liberación

```
void libera(NodoPtr l)
{
    while (l!=NULL)
    {
        NodoPtr p = l;
        l = l->sig;
        free(p);
    }
}
```

2.2 Estructuras de datos enlazadas lineales (31/34)

También se pueden declarar **estructuras de datos lineales *doblemente enlazadas***. En este tipo de estructuras, cada nodo se enlaza con el siguiente y con el elemento anterior mediante dos apuntadores, permitiendo accesos directos a ambos elementos. El siguiente tipo `NodoPtr` declara una estructura de datos lineal doblemente enlazada:

```
struct Nodo
{
    Elemento elem;
    struct Nodo * sig, * ant;
};
typedef struct Nodo* NodoPtr;
```

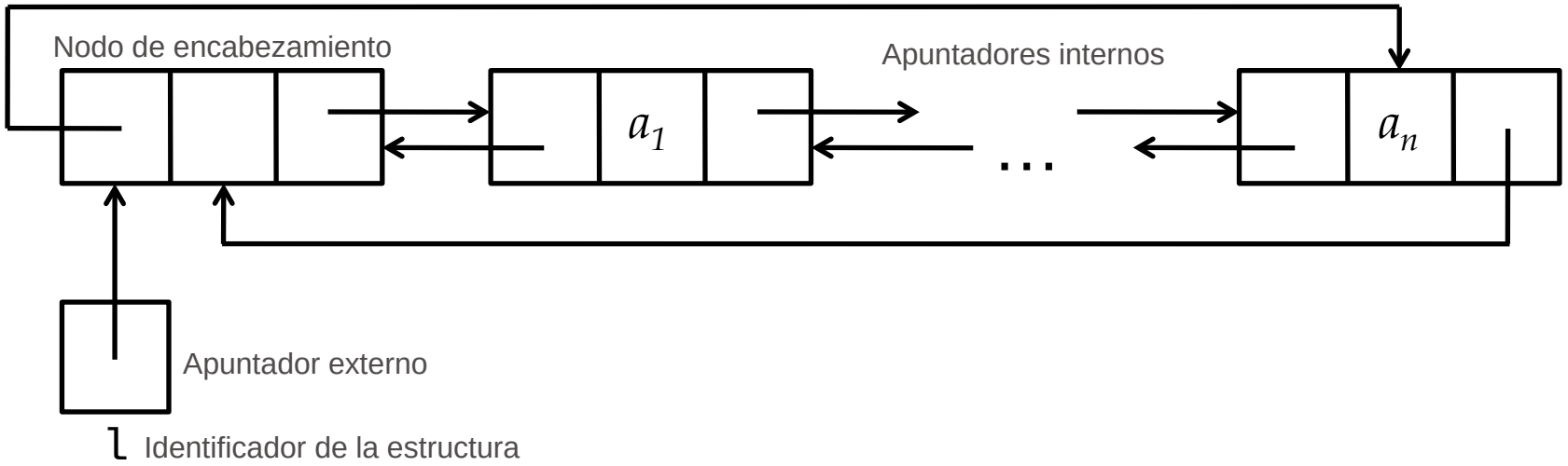
Típicamente las estructuras de datos doblemente enlazadas se construyen de forma ***circular***, es decir, el apuntador anterior del primer nodo enlaza con el último nodo, y el apuntador siguiente del último nodo enlaza con el primero.

Las estructuras de datos enlazadas lineales, bien sean con simple o con doble enlace, establecen una ***relación sucesor – predecesor*** entre sus elementos, al igual que las estructuras contiguas.

2.2 Estructuras de datos enlazadas lineales (32/34)

Representación gráfica de una estructura enlazada lineal con doble enlace circular con nodo de encabezamiento

$$l = < a_1, a_2, \ldots, a_n >$$



2.2 Estructuras de datos enlazadas lineales (33/34)

Consideraciones

- Las **inserciones** y **eliminaciones** en una estructura de datos enlazada lineal no requieren reubicación de los elementos que ya contenga para abrir o cerrar huecos, sino únicamente la **modificación de sus enlaces**, lo que hace eficientes este tipo de operaciones.

- En las estructuras enlazadas lineales, el apuntador externo puede apuntar a un **nodo de encabezamiento** enlazado con el nodo que contiene el primer elemento. El uso de nodos de encabezamiento, aunque supone un gasto de memoria adicional, permite sin embargo la codificación de operaciones homogéneas para todos los casos y por tanto más **legibles**.

2.2 Estructuras de datos enlazadas lineales (34/34)

Consideraciones

- En determinadas estructuras enlazadas lineales puede ser conveniente el uso de **apuntadores tanto al principio como al final de la estructura**.

- Típicamente las estructuras de datos doblemente enlazadas se construyen de forma circular con un único apuntador externo y un nodo de encabezamiento. El apuntador anterior del nodo de encabezamiento enlaza al último nodo, y el apuntador siguiente del último nodo enlaza con el nodo de encabezamiento.

- En una representación con simple enlace, una eliminación o inserción requiere situar un **apuntador al nodo inmediatamente anterior** del que se desea eliminar o en el que se quiere insertar. Esto no es un requerimiento en las representaciones con doble enlace.

2.3 Ejercicios resueltos (1/23)

Objetivos:

- Definir estructuras de datos enlazadas lineales.
- Comprender la necesidad del uso de una celda de encabezamiento en la implementación de operaciones de creación y manipulación de estructuras de datos enlazadas lineales.
- Crear y liberar estructuras de datos enlazadas lineales.
- Insertar, suprimir y modificar elementos en cualquier tipo de operación sobre estructuras de datos enlazadas lineales.
- Buscar elementos para cualquier tipo de operación en estructuras de datos enlazadas lineales.
- Implementar TDAs con estructuras de datos enlazadas lineales.

2.3 Ejercicios resueltos (2/23)

Dadas las siguientes definiciones de tipos:

```
struct Nodo
{
    int elem;
    struct Nodo * sig;
};
typedef struct Nodo * NodoPtr;
```

1) Implementar en *C* la función `inserta` la cual inserta el elemento `e1` antes de cualquier ocurrencia del elemento `e2` en la estructura enlazada `l`.

2) Implementar en *C* la función `suprimemin` la cual suprime el menor elemento de la estructura enlazada `l`.

2.3 Ejercicios resueltos (3/23)

```
void inserta(NodoPtr l, int e1, int e2)
{
    while(l->sig!=NULL)
    {
        if (l->sig->elem==e2)
        {
            NodoPtr nuevo = malloc(sizeof(struct Nodo));
            nuevo->elem = e1;
            nuevo->sig = l->sig;
            l->sig = nuevo;
            l = l->sig;
        }
        l = l->sig;
    }
}
```

2.3 Ejercicios resueltos (4/23)

```
void suprimemin(NodoPtr l)
{
    if (l->sig!=NULL)
    {
        NodoPtr nMin = l;
        l = l->sig;
        while(l->sig!=NULL)
        {
            if (l->sig->elem < nMin->sig->elem)
                nMin = l;
            l = l->sig;
        }
        NodoPtr eliminado = nMin->sig;
        nMin->sig = eliminado->sig;
        free(eliminado);
    }
}
```

2.3 Ejercicios resueltos (5/23)

3) Implementar *C* la función `insertaOrdenadoCreciente` la cual inserta el elemento `e`, de forma ordenada, en la estructura enlazada `l`, la cual está ordenada crecientemente (como requerimiento). Si el elemento `e` ya existe en la estructura, no se inserta.

4) Implementar *C* la función `estructuraToArray` la cual devuelve un nuevo array con los elementos copiados de la estructura enlazada `l`.

5) Implementar *C* la función `arrayToEstructura` la cual devuelve una nueva estructura enlazada con los elementos copiados del array `a` de `n` elementos.

2.3 Ejercicios resueltos (6/23)

```
void insertaOrdenadoCreciente(NodoPtr l, int e)
{
    while((l->sig!=NULL)&&(l->sig->elem<e))
        l = l->sig;
    if ((l->sig==NULL)||(l->sig->elem>e))
    {
        NodoPtr nuevo = malloc(sizeof(struct Nodo));
        nuevo->elem = e;
        nuevo->sig = l->sig;
        l->sig = nuevo;
    }
}
```

2.3 Ejercicios resueltos (7/23)

```
int * estructuraToArray(NodoPtr l)
{
    int n=0;
    for(NodoPtr aux = l->sig; aux!=NULL; aux = aux->sig)
        n++;
    int * a = malloc(n*sizeof(int));
    int i=0;
    for(NodoPtr aux = l->sig; aux!=NULL; aux = aux->sig)
    {
        a[i] = aux->elem;
        i++;
    }
    return a;
}
```

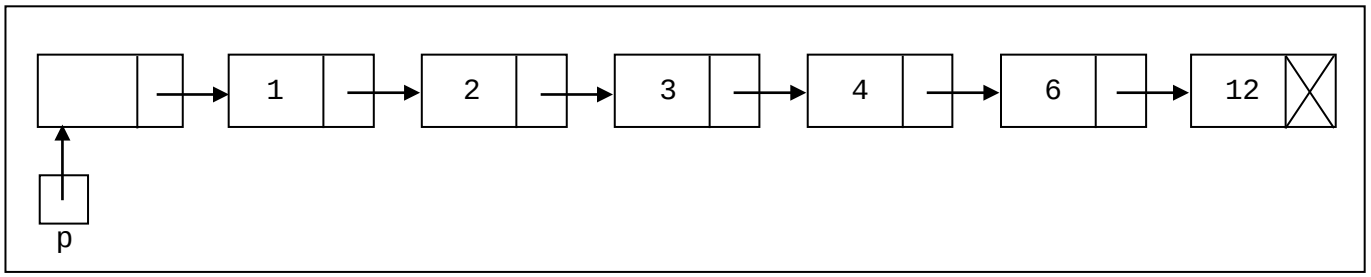
2.3 Ejercicios resueltos (8/23)

```
NodoPtr arrayToEstructura(int * a, int n)
{
     NodoPtr l = malloc(sizeof(struct Nodo));
     NodoPtr aux = l;
     for(int i=0; i<n; i++)
     {
          aux->sig = malloc(sizeof(struct Nodo));
          aux = aux->sig;
          aux->elem = a[i];
     }
     aux->sig=NULL;
     return l;
}
```

2.3 Ejercicios resueltos (9/23)

6) Implementar en *C* la función `factores` que devuelve una estructura enlazada con todos los factores de n, n>0. Se podrá hacer su uso de la función `factor`, la cual se asume implementada.

```
// Devuelve 1 (cierto) si a es factor de b y 0 (falso) en caso contrario.
int factor(int a, int b);
```



Ejemplo de estructura enlazada obtenida con `p=factores(12)`.

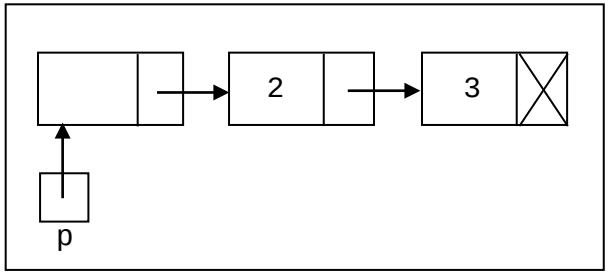
2.3 Ejercicios resueltos (10/23)

```
NodoPtr factores(int n)
{
     NodoPtr l = malloc(sizeof(struct Nodo));
     l->sig = NULL;
     for(int i=n; i>=1; i--)
     {
          if (factor(i,n))
          {
               NodoPtr nuevo = malloc(sizeof(struct Nodo));
               nuevo->elem = i;
               nuevo->sig = l->sig;
               l->sig = nuevo;
          }
     }
     return l;
}
```

2.3 Ejercicios resueltos (11/23)

7) Implementar en *C* la función `suprimeNoPrimos` que suprime los nodos que no contengan un número primo en la estructura enlazada `p`. Se podrá hacer su uso de la función `primo`, la cual se asume implementada.

```
// Devuelve 1 (cierto) si n es un número primo y 0 (falso) en caso contrario.
int primo(int n);
```



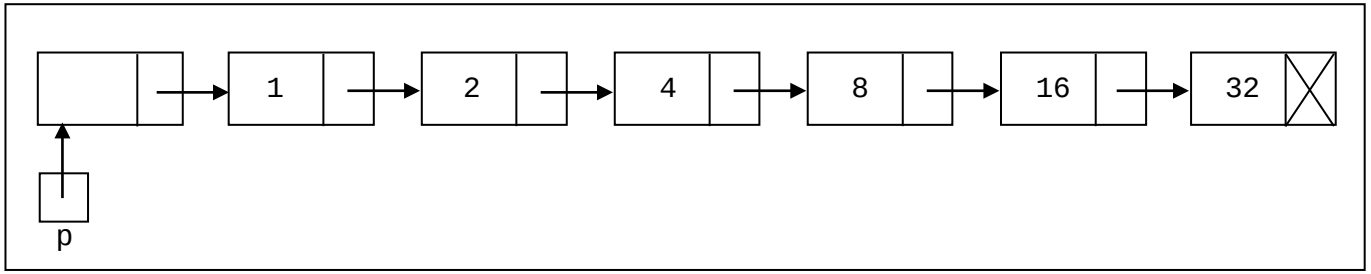
Ejemplo de estructura enlazada obtenida con `suprimeNoPrimos(p)` en la estructura del ejercicio 6.

2.3 Ejercicios resueltos (12/23)

```
void suprimeNoPrimos(NodoPtr p)
{
    while(p->sig!=NULL) {
        if (!primo(p->sig->elem))
        {
            NodoPtr eliminado = p->sig;
            p->sig = eliminado->sig;
            free(eliminado);
        }
        else p = p->sig;
    }
}
```

2.3 EJERCICIOS RESUELTOS (13/23)

8) Implementar en *C* la función `potenciasDe2` que devuelve una estructura enlazada con las `n` primeras potencias de 2, `n>0` (desde 2^0 hasta 2^{n-1}).



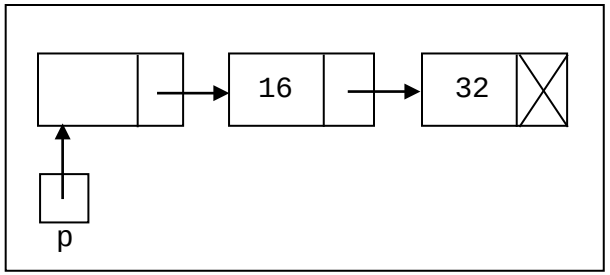
Ejemplo de estructura enlazada obtenida con `p=potenciasDe2(6)`.

2.3 Ejercicios resueltos (14/23)

```
#include <stdlib.h>
#include <math.h>
NodoPtr potenciasDe2(int n)
{
     NodoPtr l = malloc(sizeof(struct Nodo));
     NodoPtr aux = l;
     for (int i=0; i<n; i++)
     {
          aux->sig = malloc(sizeof(struct Nodo));
          aux = aux->sig;
          aux->elem = pow(2,i);
     }
     aux->sig = NULL;
     return l;
}
```

2.3 EJERCICIOS RESUELTOS (15/23)

9) Implementar en *C* la función `suprimeMenores` que suprime los nodos que contienen elementos menores que `n` en la estructura enlazada ordenada `p`.



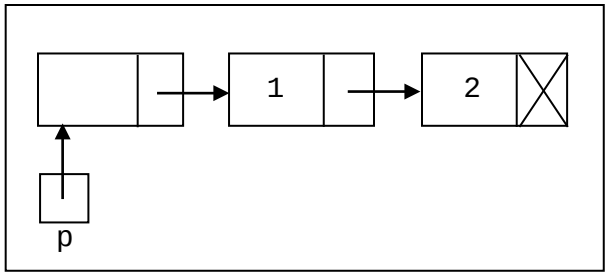
Ejemplo de estructura enlazada obtenida con `suprimeMenores(p,10)` en la estructura del ejercicio 8.

2.3 Ejercicios resueltos (16/23)

```
void suprimeMenores(NodoPtr p, int n)
{
     while (p->sig != NULL && p->sig->elem < n)
     {
          NodoPtr eliminado = p->sig;
          p->sig = eliminado->sig;
          free(eliminado);
     }
}
```

2.3 EJERCICIOS RESUELTOS (17/23)

10) Implementar en *C* la función `suprimeMayores` que suprime los nodos que contienen elementos mayores que `n` en la estructura enlazada ordenada `p`.



Ejemplo de estructura enlazada obtenida con `suprimeMayores(p,2)` en la estructura del ejercicio 8.

2.3 Ejercicios resueltos (18/23)

```
void suprimeMayores(NodoPtr p, int n)
{
    while (p->sig != NULL && p->sig->elem <= n)
        p = p->sig;
    while (p->sig != NULL)
    {
        NodoPtr eliminado = p->sig;
        p->sig = eliminado->sig;
        free(eliminado);
    }
}
```

2.3 Ejercicios resueltos (19/23)

11) Implementar *C* la función el TDA Agenda (en un fichero `Agenda.c`) sujeto a la especificación y fichero cabecera (`Agenda.h`) adjuntos.

El TDA Agenda debe implementarse con una representación enlazada (simple enlace con celda de encabezamiento) ordenada por orden lexicográfico de los nombres de los contactos (la función `nuevoContacto` debe realizar la inserción en orden).

Se asume que los nombres de los contactos no contienen más de 40 caracteres y que los teléfonos no contendrán más de 20 caracteres. Se pueden utilizar las funciones `strcmp` y `strcpy` de la biblioteca `string.h`.

2.3 Ejercicios resueltos (20/23)

```
#ifndef __AGENDA_H
#define __AGENDA_H
// El TDA Agenda define una agenda de contactos telefónicos.
typedef struct AgendaRep * Agenda;
// Crea y devuelve una nueva agenda vacía.
Agenda AgendaCrea();
// Libera la agenda a.
void AgendaLibera(Agenda a);
// Añade un nuevo contacto con nombre y telefono a la agenda a.
// Precondición: No existe un contacto asociado a nombre en la agenda a.
void AgendaNuevoContacto(Agenda a, char * nombre, char * telefono);
// Elimina el contacto asociado a nombre de la agenda a.
// Precondición: Existe un contacto asociado a nombre en la agenda a.
void AgendaEliminarContacto(Agenda a, char * nombre);
// Devuelve el teléfono asociado a nombre en la agenda a.
// Si no existe un contacto asociado a nombre en la agenda a, devuelve NULL.
char * AgendaBuscarContacto(Agenda a, char * nombre);
// Imprime los nombres y teléfonos de los contactos de la agenda a por orden
// lexicográfico de los nombres.
char * AgendaToString(char * s, Agenda a);
#endif
```

2.3 Ejercicios resueltos (21/23)

```
// Fichero Agenda.c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "Agenda.h"
#define STRING_LONG 100
struct AgendaRep
{
    char nombre[40], telefono[20];
    Agenda sig;
};
Agenda AgendaCrea()
{
    Agenda a = malloc(sizeof(struct AgendaRep));
    a->sig = NULL;
    return a;
}
```

2.3 Ejercicios resueltos (22/23)

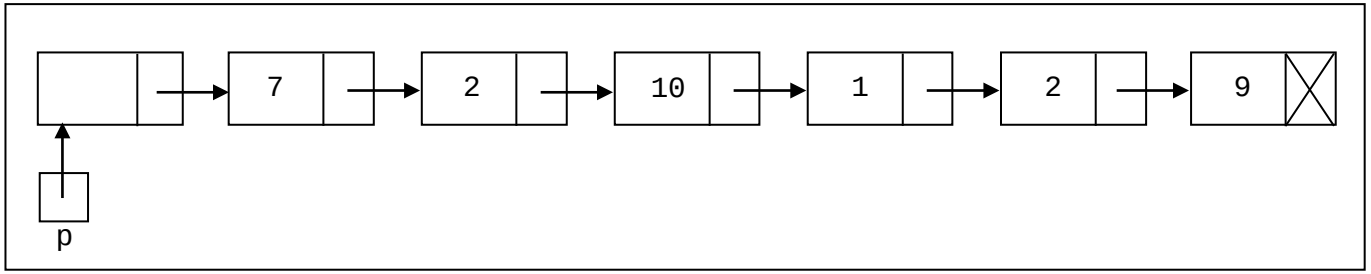
```
void AgendaLibera(Agenda a)
{
     while(a!=NULL)
     {
          Agenda aux = a;
          a = a->sig;
          free(aux);
     }
}
void AgendaNuevoContacto(Agenda a, char * nombre, char * telefono)
{
     while((a->sig!=NULL)&&(strcmp(a->sig->nombre,nombre)<0))
         a = a->sig;
     Agenda nuevo = malloc(sizeof(struct AgendaRep));
     strcpy(nuevo->nombre,nombre);
     strcpy(nuevo->telefono,telefono);
     nuevo->sig = a->sig;
     a->sig = nuevo;
}
```

2.3 Ejercicios resueltos (23/23)

```
void AgendaEliminarContacto(Agenda a, char * nombre)
{
     while(strcmp(a->sig->nombre,nombre)!=0) a = a->sig;
     Agenda eliminado = a->sig;
     a->sig = eliminado->sig;
     free(eliminado);
}
char * AgendaBuscarContacto(Agenda a, char * nombre)
{
     while((a->sig!=NULL)&&(strcmp(a->sig->nombre,nombre)<0)) a = a->sig;
     if ((a->sig==NULL)||(strcmp(a->sig->nombre,nombre)>0)) return NULL;
     return a->sig->telefono;
}
char * AgendaToString(char * s, Agenda a)
{
    char aux[STRING_LONG];
    for(; a->sig!=NULL; a=a->sig){
        sprintf(aux,"%s %s\n",a->sig->nombre,a->sig->telefono);
        strcat(s,aux);
    }
    return s;
}
```

2.4 EJERCICIOS PROPUESTOS (1/5)

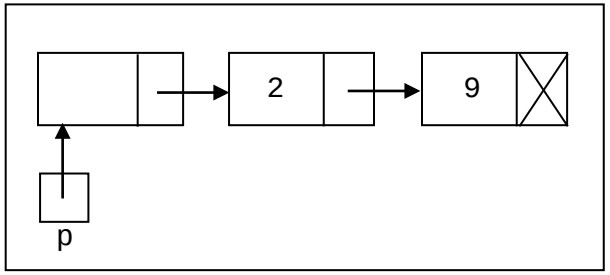
1) Implementar en *C* la función `random` que devuelve una estructura enlazada de `n` números enteros aleatorios entre `a` y `b`, con `n≥0`, `a≤b`. La expresión `a+rand()%(b-a+1)` devuelve un número entero aleatorio entre `a` y `b`, donde `a` y `b` son números enteros. Si `n=0` crea una estructura que contiene únicamente el nodo de encabezamiento con `sig=NULL`.



Ejemplo de estructura enlazada obtenida con `p=random(6,1,10)`, donde `n=6`, `a=1` y `b=10`.

2.4 EJERCICIOS PROPUESTOS (2/5)

2) Implementar en *C* la función `suprimePrimeros` que suprime los `n` primeros nodos de la estructura enlazada `p`. Si `n` es mayor o igual que el número de elementos en la estructura enlazada `p`, se eliminan todos los elementos de la estructura (excepto el nodo de encabezamiento). Si `n≤0` no se elimina ningún elemento de la estructura enlazada.



Ejemplo de estructura enlazada obtenida con `suprimePrimeros(p,4)` en la estructura del ejercicio propuesto 1.

2.4 Ejercicios propuestos (3/5)

3) Implementar en *C* la función `imprimeInverso` que imprime en pantalla los elementos de una estructura enlazada `p` en orden inverso. El tiempo de ejecución debe ser *O(n)*, por lo que se usará un array como estructura de datos auxiliar, siguiendo los siguientes pasos:

   1. Contar el número `n` de elementos de la estructura de datos `p`.
   2. Construir un array `v` de `n` elementos.
   3. Transferir al array `v` los elementos de la estructura enlazada `p`.
   4. Recorrer el array `v` en orden inverso para imprimir sus elementos en pantalla.

   Para la estructura enlazada `p` del ejemplo del ejercicio propuesto 1, `imprimeInverso(p)` imprime en pantalla `9 2 1 10 2 7`.

2.4 Ejercicios propuestos (4/5)

4) Implementar en *C* el TDA Polinomio (en un fichero `Polinomio.c`) sujeto a la especificación y fichero cabecera (`Polinomio.h`) adjuntos, teniendo en cuenta las siguientes consideraciones:
   - Debe utilizarse una representación enlazada con simple enlace ordenada decrecientemente según los exponentes. Se sugiere el uso de una celda de encabezamiento al inicio de la estructura enlazada.
   - La operación `PolinomioSuma` debe suprimir un monomio, si al sumar, su coeficiente se hace cero.

2.4 Ejercicios propuestos (5/5)

```
#ifndef __POLINOMIO_H
#define __POLINOMIO_H
// Módulo que implementa el TDA polinomio con coeficientes reales y exponentes
// enteros.
typedef struct PolinomioRep * Polinomio;
// Crea un nuevo polinomio vacío P(x) = 0
Polinomio PolinomioCrea();
// Libera el polinomio p.
void PolinomioLibera(Polinomio p);
// Suma al polinomio p un nuevo monomio con coeficiente c y grado g
void PolinomioSuma(Polinomio p, double c, int g);
// Imprime por pantalla el polinomio p decrecientemente según los exponentes
// con el siguiente formato: +2.0 x^3 +4.6 x^1 -3.2 x^0
void PolinomioImprime(Polinomio p);
// Evalúa el polinomio p en el valor x.
float PolinomioEvalua(Polinomio p, double x);
#endif // __POLINOMIO_H
```