Problemas resueltos de

LISTAS

Mónica Carreño, Andrés Sandoval, Italia Estrada
Jaime Suárez, Elvia Aispuro

UABCS

Problemas resueltos de Listas

Problemas resueltos de Listas

Mónica Carreño • Andrés Sandoval • Italia Estrada
Jaime Suárez • Elvia Aispuro

005.73
P962

Problemas resueltos de listas / Mónica Adriana Carreño León… [et al.] – México : UABCS, 2010. 322 P. : il. ; 28 cm.
ISBN: 978-607-7777-15-1
En portada: Andrés Sandoval, Italia Estrada, Jaime Suárez, Elvia Aispuro.

1.Estructura de datos (computadoras) – Problemas. 2.Programación. 3.Procesamiento electrónico de datos. I. Sandoval Bringas, Jesús Andrés, coaut. II. Estrada Cota, Italia, coaut. III. Suárez Villavicencio, Jaime, coaut. IV. Aispuro Félix, Elvia.

Primera edición, 2012

ISBN: 978-607-7777-15-1

Diseño de forros: M.D.G. Ecatl Alam López Jiménez

Impreso y hecho en México

[Prefacio]

El objetivo primordial del presente libro es servir como material de apoyo para el curso Estructura de Datos que se imparte en la UABCS (Universidad Autónoma de Baja California Sur). Dicho curso es llevado por los alumnos de las carreras LC (Licenciatura en Computación) e ITC (Ingeniería en Tecnología Computacional). No obstante, la obra también puede ser utilizada por quienes se interesen en aprender la utilización de listas ligadas.

Para un mejor entendimiento del material de este libro, es recomendable tener conocimientos de metodologías de la programación y algún lenguaje de programación (en particular C++).

La obra se centra en el planteamiento y la resolución de problemas de listas ligadas. Con la presente obra, se pretende ofrecer a aquellas personas con interés por las listas ligadas un buen número de problemas resueltos. Con detalle, explicando no sólo el cómo, sino también el porqué se resuelven así.

La obra se encuentra dividida en seis capítulos:

Capítulo 1. *Introducción a las listas*. En este capítulo se presenta un escueto resumen teórico donde se recogen las definiciones de estructuras de datos, incluyendo a las listas ligadas.

Capítulo 2. *Listas Sencillas Lineales*. En este capítulo se presenta el planteamiento de una serie de problemas resueltos utilizando listas sencillas lineales.

Capítulo 3. *Listas Sencillas Circulares*. En este capítulo se presenta el planteamiento de una serie de problemas resueltos utilizando listas sencillas circulares.

Capítulo 4. *Listas Dobles Lineales*. En este capítulo se presenta el planteamiento de una serie de problemas resueltos utilizando listas dobles lineales.

Capítulo 5. *Listas Dobles Circulares*. En este capítulo se presenta el planteamiento de una serie de problemas resueltos utilizando listas dobles circulares.

Capítulo 6. *Listas Ortogonales*. En este capítulo se presenta el planteamiento de una serie de problemas resueltos utilizando listas ortogonales.

En resumen, se considera que una serie de problemas resueltos de listas ligadas puede ser de gran utilidad para aquellas personas que se encuentren trabajando con el diseño e implementación de las operaciones básicas de las estructuras de datos.

[Capítulo]

1

Introducción a las Listas

1 INTRODUCCIÓN

1.1 Abstracción

Es un proceso mental mediante el cual el ser humano tiene la capacidad de extraer los rasgos esenciales de “algo” para representarlos por medio de un lenguaje gráfico o escrito. Esta acción subjetiva y creativa, depende del contexto psicológico de la persona que la realiza.

La abstracción debe convertirse en una habilidad para quien estudie una carrera relacionada con la computación. La capacidad de modelar una realidad por medio de herramientas computacionales requiere necesariamente de hacer continuas abstracciones, por lo que es vital conocer metodologías que desarrollen esta habilidad.

1.2 Estructuras de datos

Una estructura de datos, en general se puede definir como cualquier colección o grupo de datos organizados de tal forma que tengan asociados un conjunto de operaciones para poder manipularlos.

Las estructuras de datos se implementan a través de los lenguajes de programación y son un modelo que se caracteriza por permitir el almacenamiento y utilizar una determinada organización de datos.

Una estructura de datos puede ser de dos tipos:

- **Estructuras de datos estáticas**. Son aquellas en las que se asigna una cantidad fija de memoria y no cambia durante la ejecución de un programa, es decir, las variables no pueden crearse ni destruirse durante la ejecución del programa.

- **Estructuras de datos dinámicas**. Son aquellas en las que su ocupación en memoria puede aumentar o disminuir durante el tiempo de ejecución de un programa.

  A su vez las estructuras de datos dinámicas se pueden clasificar en lineales y no lineales:

  - **Estructuras lineales**. Son aquellas en las que se definen secuencias como conjuntos de elementos entre los que se establece una relación de predecesor y sucesor. Las diferentes estructuras de datos basadas

en este concepto se diferencian por las operaciones de acceso a los elementos y manipulación de la estructuras. Existen tres estructuras lineales especialmente importantes: las pilas, las colas y las listas.

- **Estructuras no lineales**. Son aquellas en las que no existe una relación de adyacencia entre sus elementos, es decir, un elemento puede estar relacionado con cero, uno o más elementos. Existen dos estructuras no lineales especialmente importantes: los árboles y los grafos.

1.3 Listas ligadas

Las listas ligadas forman parte del grupo de las estructuras lineales dinámicas. Cada lista está conformada por un conjunto de elementos ($x_{1,}x_2...x_{n}$) relacionados de forma que el elemento x_{k+1} sigue al elementos x_{k} *para* $1 \leq k \leq n$. La relación de secuencia puede ser de tipo físico o de tipo lógico.

La lista ligada es una de las estructuras usadas con mayor frecuencia en el manejo de información, según el tipo de aplicación sus elementos reciben diferentes nombres. Sus elementos se denominan nodos, y normalmente todos tienen la misma conformación. Los nodos se encuentran dispersos en el medio de almacenamiento, pero cada uno de ellos contiene un apuntador que almacena la dirección de memoria del siguiente nodo. La lista tiene un punto de entrada que puede ser el primer nodo perteneciente a la lista o, en algunos casos, se inserta un nodo adicional denominado cabecera. En cualquiera de los dos casos es necesario guardar en un apuntador la dirección de entrada a la lista para acceder a ella.

El número de elementos de una lista se llama longitud. Si la lista tiene 0 elementos se denomina lista vacía y se representa con el valor nulo.

Según la conformación del nodo y el número de ligas, las listas se pueden clasificar en:

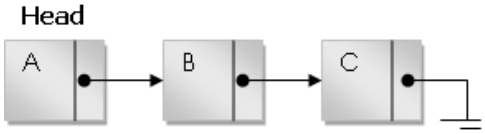
- Listas sencillas
- Listas dobles
- Listas de listas (Multiencadenamiento)

1.3.1 Listas sencillas

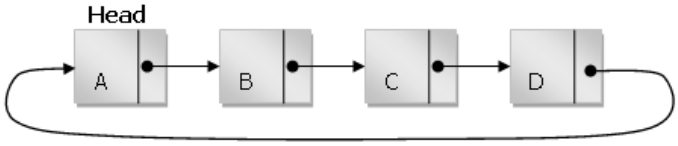
Una lista sencilla es aquella en la que sus nodos se encuentran enlazados únicamente por una liga, es decir, cada nodo apunta al siguiente nodo de la lista y cada nodo es apuntado por el nodo anterior de lista, a excepción del primer nodo y del último nodo de la lista. Una lista sencilla puede ser implementada como lineal

o circular. En una lista lineal el último nodo de la lista apunta hacia un valor nulo, mientras que en una lista circular el último nodo de la lista apunta hacia el primer nodo de la lista.

Lista Sencilla Lineal



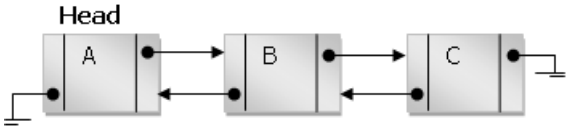
Lista Sencilla Circular



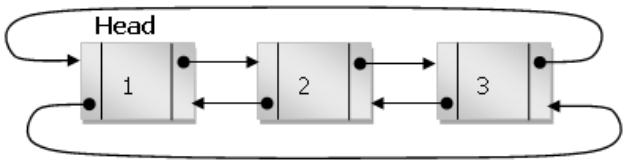
1.3.2 Listas dobles

Una lista doble es aquella en la que sus nodos se encuentran encadenados por dos ligas, es decir, cada nodo apunta al siguiente nodo de la lista, así como al nodo que le antecede en la lista. Una lista doble puede ser implementada como lineal o circular. En una lista lineal, la liga siguiente del último nodo y la liga anterior del primer nodo apuntan hacia un valor nulo, mientras que en una lista circular la liga siguiente del último nodo apunta hacia el primer nodo de la lista y la liga anterior del primer nodo apunta hacia el último nodo de la lista.

Lista Doble Lineal

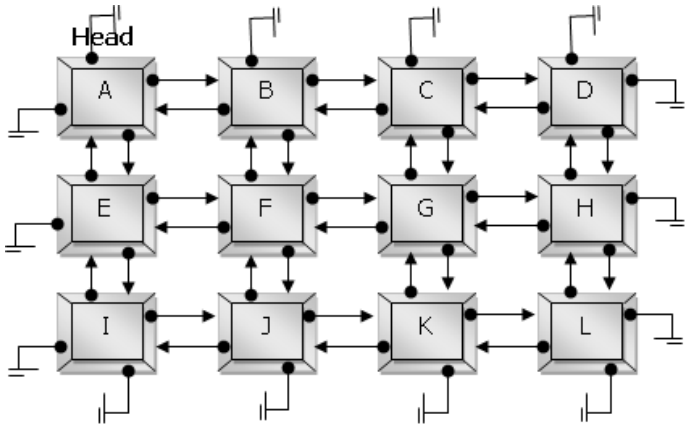


Lista Doble Circular



1.3.3 Listas ortogonales

Una lista ortogonal es aquella en la que sus nodos se encuentran encadenados por cuatro ligas, es decir, cada nodo se encuentra doblemente ligado en forma horizontal, y cada nodo se encuentra doblemente ligado en forma vertical. Una lista ortogonal puede ser implementada como lineal o circular. Este tipo de listas se puede utilizar para representar matrices.



1.3.4 Operaciones básicas

En una lista se pueden efectuar operaciones por medio de algoritmos que se deben desarrollar de acuerdo con el tipo de lista. Algunas de las operaciones básicas que se pueden efectuar sobre una lista son:

1) **Recorrido.** Esta operación consiste en visitar todos los nodos que forman parte de una lista. Para recorrer todos los nodos de la lista es necesario posicionarse en el primer nodo de la lista y después avanzar hacia el nodo que apunte la liga siguiente y así sucesivamente hasta encontrar el fin de la lista.

2) **Inserción.** Esta operación consiste en agregar un nuevo nodo a una lista. La ubicación del nuevo nodo puede ser al inicio, al final o en cualquier posición dentro de la lista.

3) **Borrado.** Esta operación consiste en eliminar un nodo de la lista y redireccionar las ligas de los nodos antecesor y sucesor para el caso de un nodo que se encuentre en una posición intermedia. El borrado también se puede aplicar tanto al primer nodo de la lista como al último nodo de la lista.

4) **Búsqueda.** Esta operación consiste en recorrer todos los nodos de la lista desde el primer nodo para ir comparando el valor de cada nodo con el valor

que se esta buscando hasta encontrar el nodo con el valor indicado o encontrar el fin de la lista.

1.3.5 Ventajas y desventajas

Las listas son estructuras de datos que son dinámicas, esto significa que adquieren espacio y liberan espacio a medida que se necesita. Son muy versátiles, pueden definirse estructuras más complejas a partir de las listas, como por ejemplo arreglos de listas. En algunas ocasiones los grafos se definen como listas de adyacencia. También se utilizan para las tablas de hash (dispersión) como arreglos de listas.

Son eficaces para diseñar colas de prioridad, pilas y colas sin prioridad, y en general cualquier estructura cuyo acceso a sus elementos se realice de manera secuencial.

Sin embargo, hay una advertencia. Como regla general siempre hay que tener cuidado al manejar direcciones de espacios de memoria, porque es posible que se acceda a una localidad de memoria de la cual no se deseaba cambiar su contenido.

1.4 Nomenclatura

Un nodo en una lista ligada tiene la siguiente conformación:

Dirección	Es su ubicación en la memoria (principal o secundaria).
Dato o Información	Son los campos donde se almacena la información, o el elemento que hace parte de la lista.
Enlace o Liga	Es un campo que permite almacenar la dirección del siguiente nodo de la lista.

Si se supone que P es un apuntador a un nodo dado, mediante la siguiente nomenclatura se puede identificar cada uno de los campos que pertenecen al nodo, y es la que se utiliza a lo largo de este libro.

p	Apuntador al nodo cuya dirección de memoria está en p.
p(dato)	El campo de información para el nodo apuntado por p.
p(liga)	El campo almacena la dirección de memoria del nodo sucesor del nodo p. Dependiendo del tipo de lista un nodo puede tener más de un enlace.
p(liga) ← NULL	El campo almacena un valor nulo para indicar que no existe sucesor del nodo p.

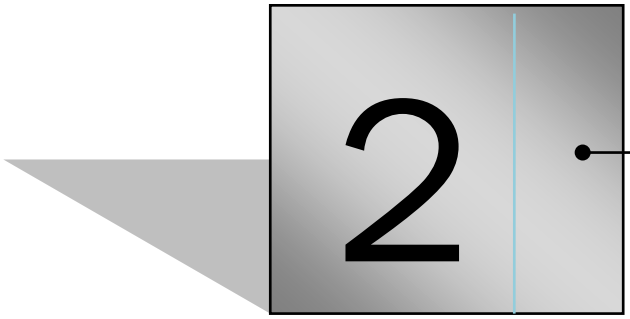
Para el diseño de los algoritmos en pseudocódigo propuestos en este libro se utiliza la siguiente nomenclatura:

Nuevo ( )	Reserva un espacio en memoria y la asigna a una variable de tipo apuntador.
Leer ( )	Permite introducir un valor por el usuario a una variable.
Eliminar ( )	Elimina el espacio de memoria apuntado por la variable de tipo apuntador.
Mensaje (‘ ’)	Visualiza el mensaje que se encuentra entre comillas.
←	Asigna un valor a una variable.
Desplegar( )	Visualiza el contenido de una variable.

Para la implementación del código utilizando el lenguaje de programación C++ de los algoritmos propuestos se utilizan las siguientes equivalencias:

Pseudocódigo	C++
Nuevo(p)	`p=nuevo();`
p(liga) ← NULL	`p->liga = NULL;`
Leer (p(dato))	`cin >> p->dato;`
Eliminar(p)	`delete(p);`
Desplegar (p(dato))	`cout << p->dato;`
p← p(liga)	`p = p->liga;`
head ← p	`head = p;`

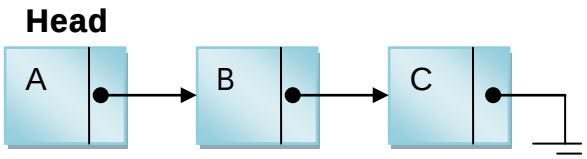
[Capítulo]



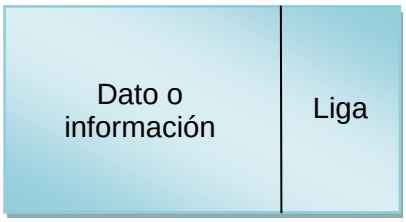
Listas Sencillas Lineales

2 LISTAS SENCILLAS LINEALES

La característica principal de una lista sencilla lineal es que la liga del último nodo apunta hacia el valor nulo.



El nodo de una lista sencilla lineal debe contener como mínimo dos campos: uno para almacenar la información y otro para guardar la dirección de memoria hacia el siguiente nodo de la lista. En la figura se puede apreciar la estructura del nodo para una lista sencilla.



Para definir la estructura del nodo en C++ se hace lo siguiente:

```
struct apuntador
{
  char dato;
  apuntador *liga;
};
```

Para simplificar la asignación de memoria se utiliza la siguiente función:

```
nodo nuevo()
{
 nodo p;
 p = new struct apuntador;
 return p;
}
```

Se presenta la clase lista_sencilla_lineal, la cual incluye la variable head y los métodos de las operaciones que se desarrollan en este capítulo para el manejo de las listas sencillas lineales.

```
class lista_sencilla_lineal
{
  nodo head;
  public:
   lista_sencilla_lineal();
   void crear();
   void desplegar();
   void mayusculas();
   int tamano();
   void insertar_final();
   void insertar_inicio();
   void insertar(int posicion);
   void borrar_ultimo();
   void borrar_inicio();
   void borrar(int posicion);
   void desplegar_invertida();
   void burbuja();
   void invertir();
   void concatenar(lista_sencilla_lineal &b);
   void eliminar_subcadena(int n, int x);
   void intercalar(lista_sencilla_lineal &a);
   void particionar(lista_sencilla_lineal &a,
        lista_sencilla_lineal &b);
   int buscar(char valor);
   void eliminar_repetidos(char valor);
   void eliminar();
   int posicion(char valor);
   int comparar(lista_sencilla_lineal a);
   void burbuja_ligas();
   void reemplazar(int pos, char *valor);
   void eliminar_subcadena(char *valor);
};
```

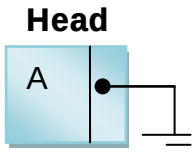
2.1 Crear una lista

Diseñar un algoritmo que permita crear una lista lineal sencilla con n número de nodos.

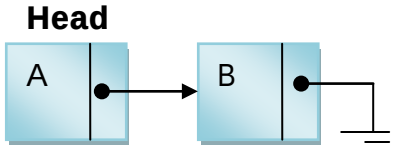
Análisis del problema

Para resolver este problema es necesario la utilización de un ciclo que estará generando cada uno de los nodos que formaran parte de la lista. Es necesario introducir la información de cada uno de los nodos dentro del ciclo.

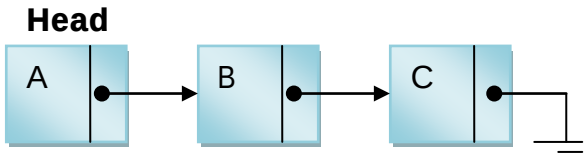
Primera iteración



Segunda iteración



Tercera Iteración



Solución en pseudocódigo

```
Repite
    Nuevo (p)
    Leer (p(dato))
    p(liga) ← null
    Si head = null entonces
        head ← p
    De lo contrario
        q(liga) ← p
    q ← p
    Leer (otro)
Hasta (otro = NO)
```

Código en C++

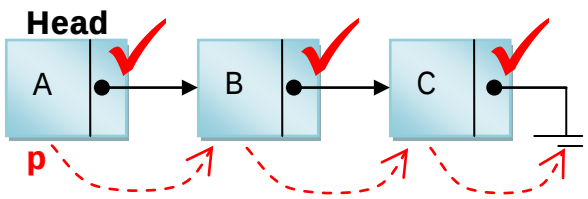
```
void lista_sencilla_lineal::crear()
{
  nodo p,q;
  char otro;
  do
  {
    p=nuevo();
    cout << "p(dato) = ";
    cin >> p->dato;
    p->liga = NULL;
    if (head == NULL)
        head = p;
    else
        q->liga = p;
    q=p;
    cout << "Capturar otro nodo s/n ? " ;
    cin >> otro;

  } while (otro == 's');
}
```

2.2 Recorrer una lista

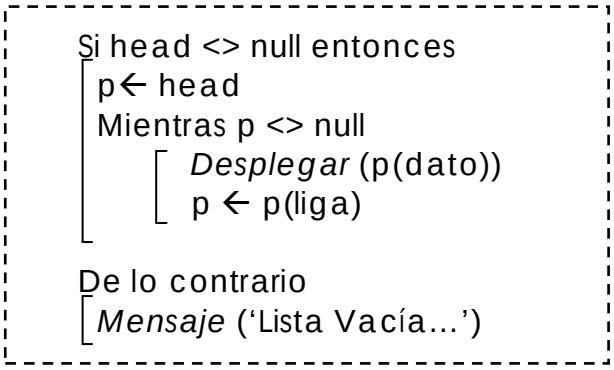
Diseñar un algoritmo que permita desplegar el contenido de todos los nodos de una lista lineal sencilla.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se recorre toda la lista desde el primer nodo donde se encuentra *head* hasta encontrar el valor de nulo, visualizando el contenido de cada uno de los nodos de la lista.



Solución en pseudocódigo



Código en C++

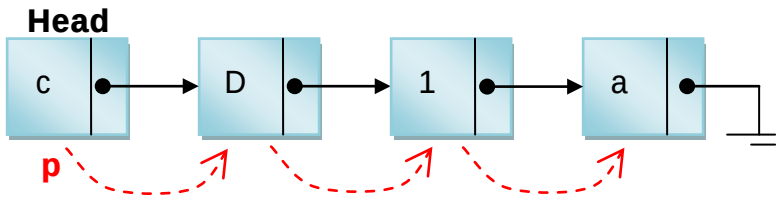
```
void
lista_sencilla_lineal::desplegar()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    while (p!=NULL)
    {
       cout << p->dato;
       p = p->liga;
    }
  }
  else
    cout <<  "Lista Vacía";
}
```

2.3 Convertir a mayúsculas

Diseñar un algoritmo que permita convertir todos los elementos alfabéticos de una lista lineal sencilla que se encuentren en minúsculas a mayúsculas.

Análisis del problema

Para resolver este problema es necesario recorrer toda la lista desde el inicio, para ir comparando el valor del nodo y en caso de que sea una letra convertirla a mayúscula.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras  p <>  null
        Si p(dato) es una letra entonces
            p(dato) ← mayúscula(p(dato))
        p ← p(liga)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

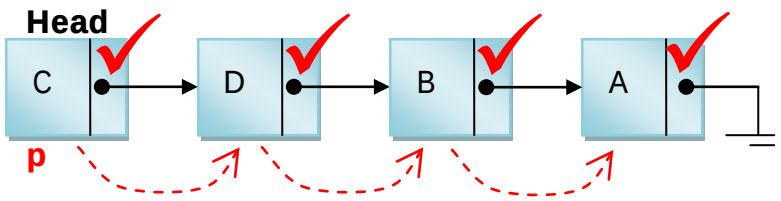
```
void lista_sencilla_lineal::mayusculas()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    while (p!=NULL)
    {
       if (p->dato>='a' && p->dato <= 'z')
            p->dato -=32;
       p = p->liga;
    }
  }
  else
    cout <<  "Lista Vacía";
}
```

2.4 Calcular tamaño

Diseñar un algoritmo que permita determinar el tamaño de una lista lineal sencilla.

Análisis del problema

Para calcular el tamaño de la lista es necesario recorrer todos los nodos de la lista desde el primer nodo hasta encontrar el valor de nulo. Para contar el total de nodos se utiliza un contador que se va incrementando.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    total ← 1
    Mientras p(liga) <> null
        p ← p(liga)
        total ← total + 1
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
int lista_sencilla_lineal::tamano()
{
  nodo p;
  int total;
  if (head != NULL)
  {
    p = head;
    total = 1;
    while (p->liga !=NULL)
    {
       p = p->liga;
       total++;
    }
  }
  else
    cout <<  "Lista Vacía";
  return total;
}
```

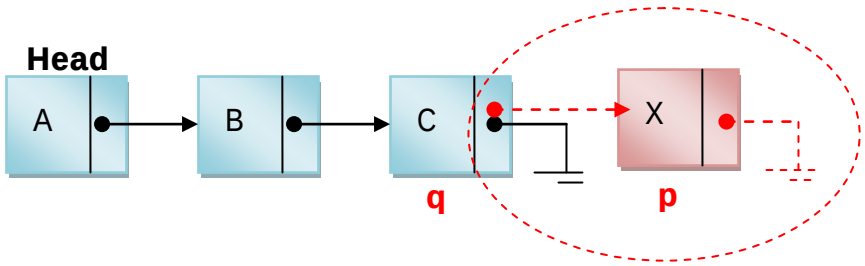
2.5 Insertar al final

Diseñar un algoritmo que permita insertar un nodo al final de una lista lineal sencilla.

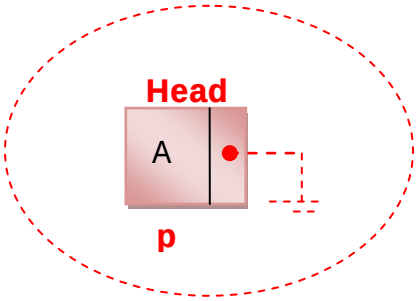
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía es necesario recorrer toda la lista para ubicarse en el último nodo, crear el nuevo nodo y ligar el nuevo nodo con el último nodo. Si la lista está vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo.

Caso 1: La lista contiene al menos un elemento.



Caso 2: La lista no contiene elementos.



Solución en pseudocódigo

```
Nuevo (p)
Leer (p(dato))
p(liga) ← null
Si head <> null entonces
    q ← head
    Mientras q(liga) <> null
        q ← q(liga)
    q(liga)← p
De lo contrario
    head ← p
```

Código en C++

```
void
lista_sencilla_lineal::insertar_final()
{
   nodo p,q;
   p = nuevo();
   cout << "p(dato) = ";
   cin >> p->dato;
   p->liga = NULL;
   if (head != NULL)
   {
        q = p;
        while (q->liga != NULL)
            q = q->liga;
        q->liga = p;
   }
   else
        head = p;
}
```

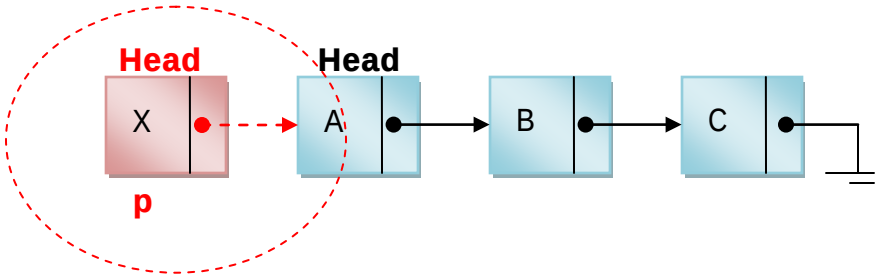
2.6 Insertar al inicio

Diseñar un algoritmo que permita insertar un nodo al inicio de una lista lineal sencilla.

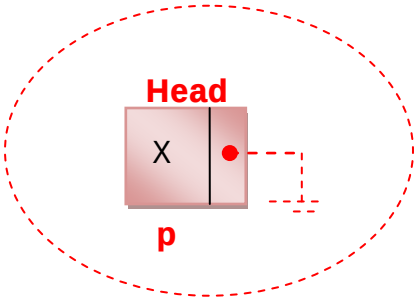
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se crea el nuevo nodo y se liga con el primero nodo de la lista y *head* se mueve al nuevo nodo. Si la lista esta vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo.

Caso 1: La lista contiene al menos un elemento.



Caso 2: La lista no contiene elementos.



Solución en pseudocódigo

```
Nuevo (p)
Leer (p(dato))
p(liga) ← head
head ← p
```

Código en C++

```
void lista_sencilla_lineal::insertar_inicio()
{
   nodo p;
   p = nuevo();
   cout << "p(dato) = ";
   cin >> p->dato;
   p->liga = head;
   head = p;
}
```

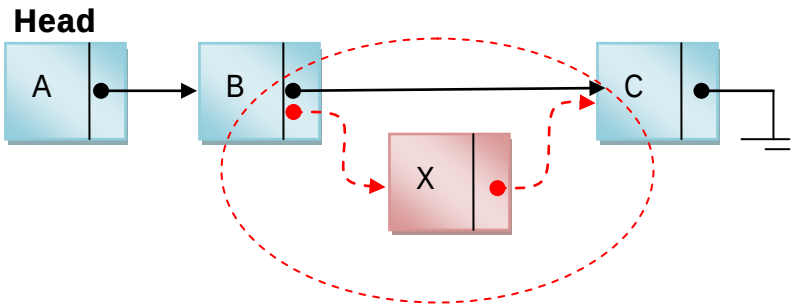
2.7 Insertar en cualquier posición

Diseñar un algoritmo que permita insertar un nodo en cualquier posición en una lista lineal sencilla.

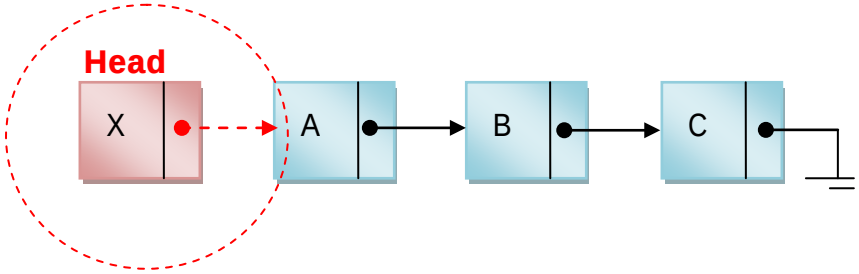
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos y si la posición en la que se desea insertar el nuevo es válida, es decir si es menor o igual al total de los nodos de la lista. Se deben consideran los casos siguientes:

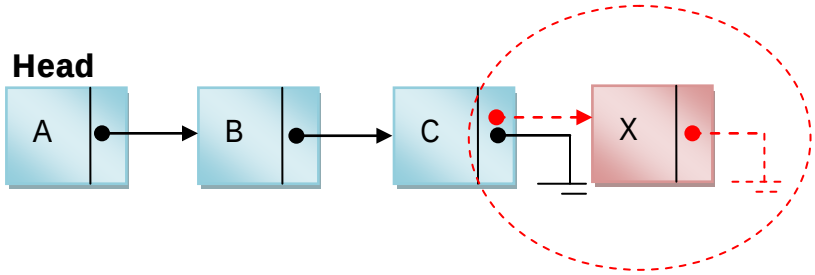
Caso 1: Insertar el nuevo nodo en una posición intermedia dentro de la lista.



Caso 2: Insertar el nuevo nodo al inicio de la lista. En este caso también se debe determinar si el nodo que se inserta es el primero de la lista, y en su caso, ubicar a head en dicho nodo.



Caso 3: Insertar el nuevo nodo al final de la lista.



Solución en pseudocódigo

```
Leer (posición)
p ← head
c ← 0
Mientras p <> null
    p ← p(liga)
    c ← c + 1
Si (posición  > 0) y (posición <= c+1) entonces
    Nuevo (p)
    Leer (p(dato))
    Si pos = 1 entonces
        p(liga) ← head
        head ← p
    De lo contrario
        q ← head
        para i = 1 hasta pos – 1
            q ← q(liga)
        p(liga) ← q(liga)
        q(liga) ← p
De lo contrario
    Mensaje ('Posición incorrecta...')
```

Código en C++

```
void lista_sencilla_lineal::insertar(int posicion)
{
   nodo p,q;
   int c,i;
   p = head;
   c = 0;
   while (p != NULL)
   {
       p = p->liga;
       c++;
   }
   if ((posicion > 0) && (posicion <= c+1))
   {
       p = nuevo();
       cout << "p(dato) = ";
       cin >> p->dato;
       if (posicion==1)
       {
           p->liga = head;
           head = p;
       }
       else
       {
           q = head;
           for(i=1; i<=posicion-1; i++)
                q = q->liga;
           p->liga = q->liga;
           q->liga = p;
       }
   }
   else
        cout <<  "Posición Incorrecta...";
}
```

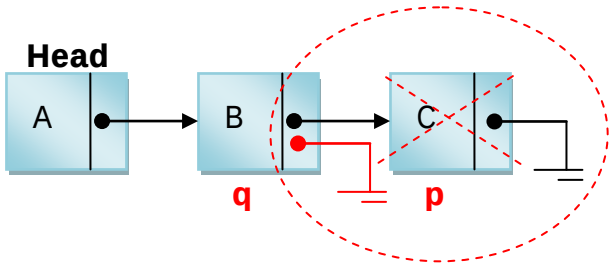
2.8 Borrar el último nodo

Diseñar un algoritmo que permita borrar el último nodo de una lista lineal sencilla.

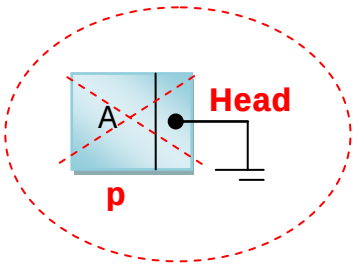
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene al menos un elemento. Es necesario recorrer los nodos de la lista para ubicarse en la penúltima posición y eliminar el último nodo. En el caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *Head* en nulo.

Caso 1: La lista contiene al menos dos elementos y se elimina el último.



Caso 2: La lista contiene un solo nodo. En este caso es necesario inicializar el valor de head a nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras (p(liga) <> null)
        q ← p
        p ← p(liga)
    Si p<>head entonces
        q(liga) ← null
    De lo contrario
        head ← null
    Eliminar (p)
De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

```
void lista_sencilla_lineal::borrar_ultimo()
{
  nodo p,q;
  if (head != NULL)
  {
    p = head;
    while (p->liga != NULL)
    {
        q = p;
        p = p->liga;
    }
    if (p != head)
        q->liga = NULL;
    else
        head = NULL;
    delete(p);
  }
  else
    cout <<  "Lista Vacía";
}
```

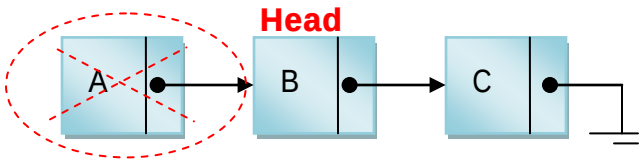
2.9 Borrar el primer nodo

Diseñar un algoritmo que permita borrar el primer nodo de una lista lineal sencilla.

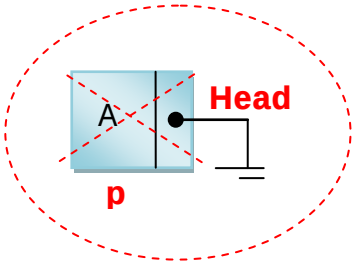
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene al menos un elemento. Si la lista contiene elementos se elimina el primer nodo de la lista y la variable *Head* se mueve al siguiente nodo de la lista. En el caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *Head* en nulo.

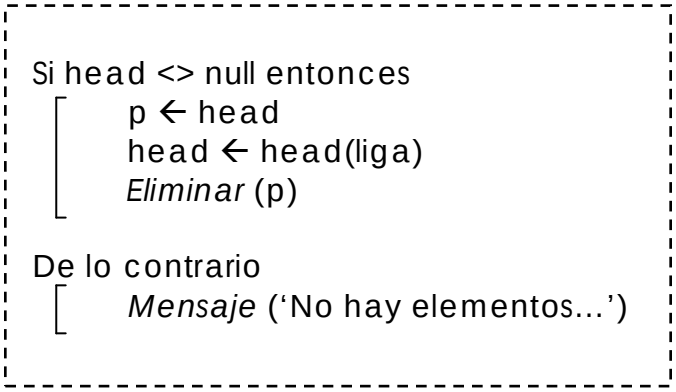
Caso 1: La lista contiene al menos dos elementos. En este caso se elimina el primero y el head debe avanzar a la siguiente posición.



Caso 2: La lista contiene un solo nodo. En este caso es necesario inicializar el valor de head a nulo.



Solución en pseudocódigo



Código en C++

```
void lista_sencilla_lineal::borrar_inicio()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    head = head->liga;
    delete(p);
  }
  else
    cout <<  "No hay elementos";
}
```

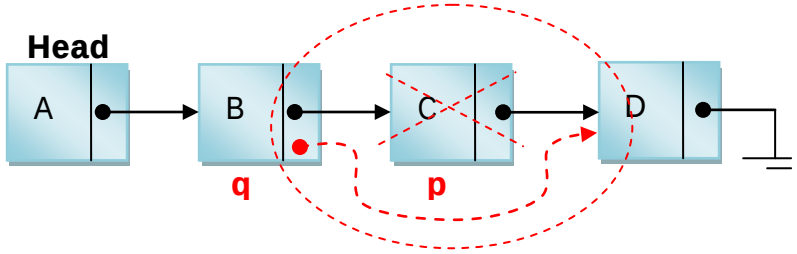
2.10 Borrar cualquier nodo

Diseñar un algoritmo que permita borrar un nodo en cualquier posición en una lista lineal sencilla.

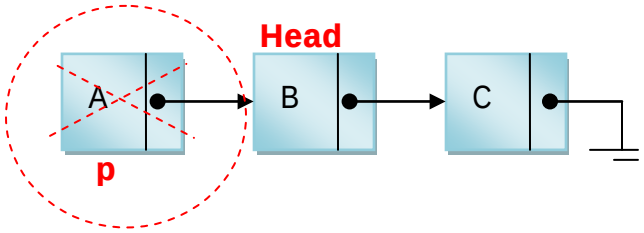
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos y la posición del elemento que se desea eliminar es válida. En el caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *Head* en nulo. Si la posición es la última o la primera se utilizan los algoritmos para eliminar el último o el primer nodo. Si la posición es intermedia es necesario ligar el nodo antecesor y el sucesor del nodo que se elimina.

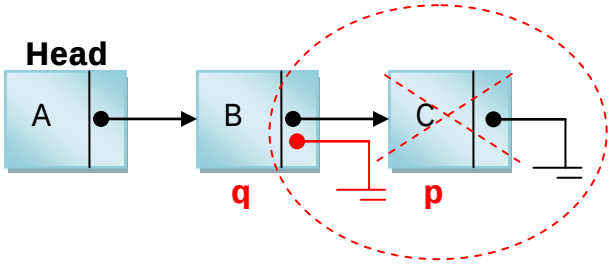
Caso 1: Borrar un nodo que se encuentre en una posición intermedia.



Caso 2: Borrar el primer nodo de la lista.



Caso 3: Borrar el último nodo de la lista.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (pos)
    p ← head
    c ← 1
    Mientras (c <> pos) y (p <> null)
        q ← p
        p ← p(liga)
        c ← c + 1
    Si c = pos entonces
        Si p = head entonces
            head ← head (liga)
        De lo contrario
            q(liga) ← p(liga)
        Eliminar (p)
    De lo contrario
        Mensaje (Posición inválida)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_sencilla_lineal::borrar(int posicion)
{
 nodo p,q;
 int c;
 if (head != NULL)
 {
   p = head;
   c = 1;
   while ((c != posicion) && (p != NULL))
   {
       q = p;
       p = p->liga;
       c++;
   }
   if (c==posicion)
   {
       if (p==head)
```

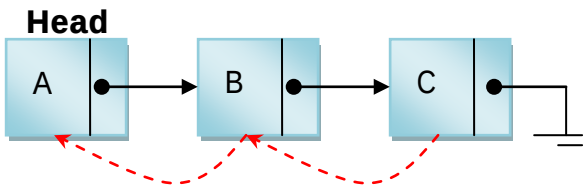
```
            head = head->liga;
        else
            q->liga = p->liga;
        delete (p);
    }
    else
        Cout <<  "Posición inválida...";
 }
 else
    cout <<  "Lista Vacía";
}
```

2.11 Desplegar invertida

Diseñar un algoritmo que permita desplegar el contenido de los nodos de una lista lineal sencilla de forma invertida.

Análisis del problema

Para resolver este problema es necesario verificar si la lista contiene elementos. Para desplegar el contenido de lista en forma inversa se tiene que ir recorriendo la lista de atrás hacia delante, pero como la lista no está ligada en esa dirección, se utilizan dos apuntadores uno se deja en la última posición y el otro en la penúltima posición. Después el apuntador que se encuentra en la última posición se ubica en el penúltimo nodo y se recorre nuevamente la lista desde el inicio dejando un apuntador un nodo antes del penúltimo, y así sucesivamente hasta llegar al primer nodo de la lista.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(liga) <> null
        p← p(liga)
    Mientras (p <> head)
        q ← head
        Mientras q(liga) <> p
            q ← q(liga)
        Desplegar (p(dato))
        p ← q
    Desplegar (p(dato))

De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

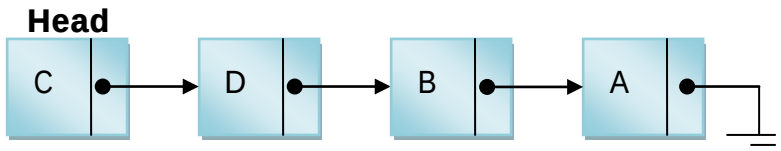
```
void lista_sencilla_lineal::desplegar_invertida()
{
   nodo p,q;
   If (head != NULL)
   {
      p = head;
      while (p->liga != NULL)
        p = p->liga;
      while (p != head)
      {
          q = head;
          while (q->liga != p)
            q = q->liga;
          cout << p->dato;
          p = q;
      }
      cout << p->dato;
   }
   else
      cout <<  "Lista Vacía";
}
```

2.12 Ordenar con método de la burbuja

Diseñar un algoritmo que permita ordenar una lista lineal sencilla utilizando el método de la burbuja.

Análisis del problema

El método de la burbuja es uno de los métodos más sencillos de ordenación. Para la implementación del método se requieren dos ciclos anidados para ir comparando los elementos y hacer los intercambios en donde sea necesario.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(liga) <> null
        q ← p(liga)
        Mientras (q <> null)
            Si (q(dato < p(dato) entonces
                aux ← p(dato)
                p(dato) ← q(dato)
                q(dato) ← aux
            q ← q(liga)
        p ← p(liga)
De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

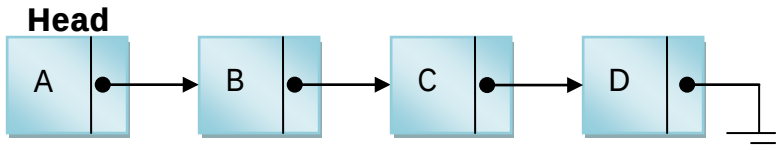
```
void lista_sencilla_lineal::burbuja()
{
  nodo p,q;
  char aux;
  if (head != NULL)
  {
      p = head;
      while (p->liga != NULL)
      {
          q = p->liga;
          while (q != NULL)
          {
              if (q->dato < p->dato)
              {
                  aux = p->dato;
                  p->dato = q->dato;
                  q->dato = aux;
              }
              q = q->liga;
          }
          p = p->liga;
      }
  }
  else
      cout <<  "No hay elementos";
}
```

2.13 Invertir la lista

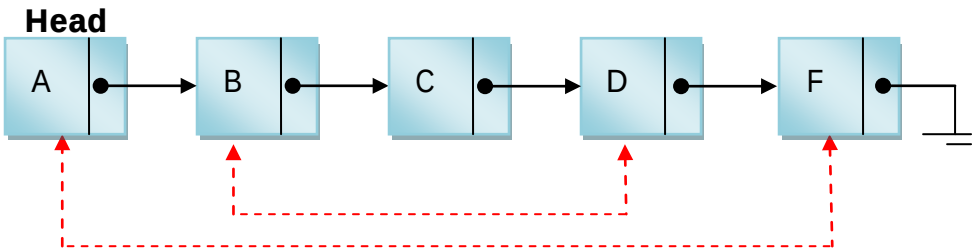
Diseñar un algoritmo que permita invertir los nodos de una lista lineal sencilla.

Análisis del problema

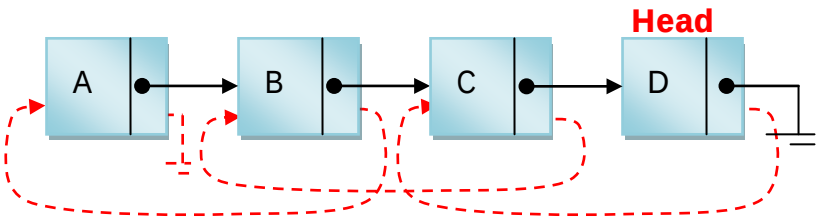
La operación de invertir una lista puede hacerse de dos maneras: la primera intercambiando únicamente los datos y la segunda dejando los datos en la posición de memoria en la que se encuentran y cambiar el ligado de los nodos para que la lista quede invertida.



Solución 1: Moviendo datos.



Solución 2: Moviendo ligas.



Solución en pseudocódigo

a) Moviendo datos

```
Si head <> null  y head(liga) <> null entonces
    p ← head
    Mientras p(liga) <> null
        p ← p(liga)
    r ← p
    Repite
        q ← head
        Mientras q(liga) <> p
            q ← q(liga)
        p(liga) ← q
        p ← q
    Hasta p = head
    p(liga) ← null
    head ← r
De lo contrario
    Mensaje ('No hay suficientes elementos...')
```

Código en C++

```
void lista_sencilla_lineal::invertir()
{
   nodo p,q,r;
   if ((head != NULL) && (head->liga != NULL))
   {
       p = head;
       while (p->liga != NULL)
           p = p->liga;
       r = p;
       do
       {
           q = head;
           while (q->liga != p)
                q = q->liga;
```

```
            p->liga = q;
            p = q;
        } while (p != head);
        p->liga = NULL;
        head = r;
    }
    else
        cout <<  "No hay suficientes elementos";
}
```

Solución en pseudocódigo

b) Moviendo ligas

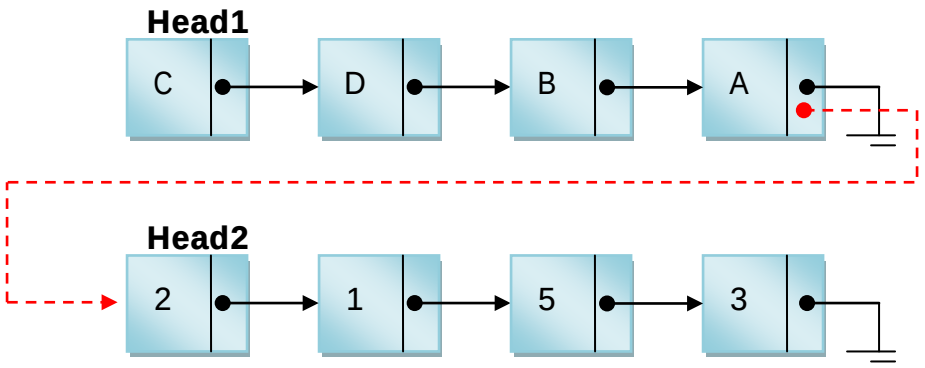
```
Si head <> null  y head(liga) <> null entonces
    p ← head
    q ← null
    Mientras head <> null
        head ← head(liga)
        p(liga) ← q
        q ← p
        p← head
    head ← q

De lo contrario
    Mensaje ('No hay suficientes elementos...')
```

2.14 Concatenar dos listas

Diseñar un algoritmo que permita concatenar los nodos de dos listas lineales sencillas.

Análisis del problema

La operación de concatenar consiste en unir dos listas como se muestra en la figura, para ello es necesario ubicarse en el último nodo de la primera lista y hacer que la liga de ese nodo apunte al primer nodo de la segunda lista y eliminar la variable *Head* de la segunda lista. Se debe validar que las dos listas contengan al menos un elemento.



Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
    p ← head1
    Mientras p(liga) <> null
        p ← p(liga)
    p(liga) ← head2

De lo contrario
    Mensaje ('No hay listas...')
```

Código en C++

```
void lista_sencilla_lineal::concatenar(
lista_sencilla_lineal &b)
{
   nodo p;
   if (head != NULL && b.head != NULL)
   {
        P = head;
        while (p->liga != NULL)
             p = p->liga;
        p->liga = b.head;
        b.head = NULL;
   }
   else
        cout <<  "No hay listas";
}
```

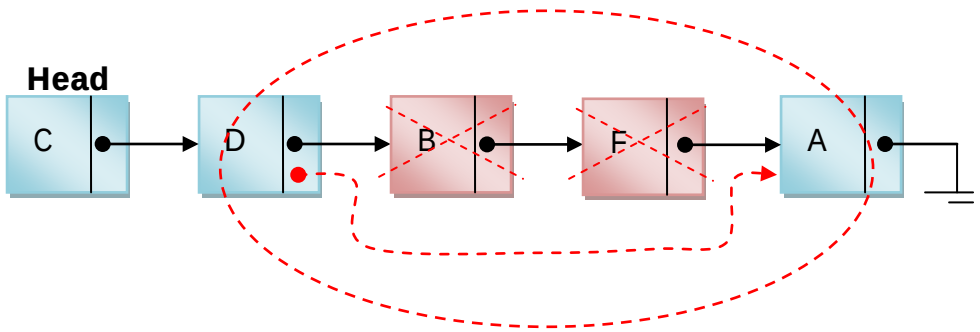
2.15 Eliminar n número de nodos

Diseñar un algoritmo que permita eliminar n número de nodos a partir de la posición x en una lista sencilla lineal.

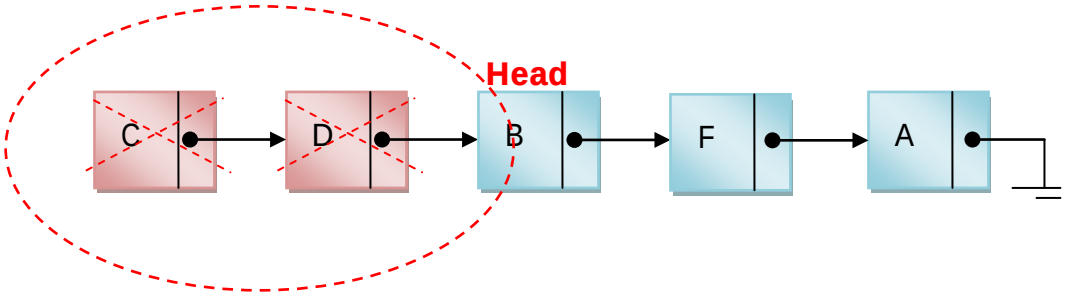
Análisis del problema

Para resolver este problema es necesario contar con el valor de n y x. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación, es decir, si existe la posición a partir de la cual se van a eliminar elementos y suficientes nodos para cumplir con el valor de n.

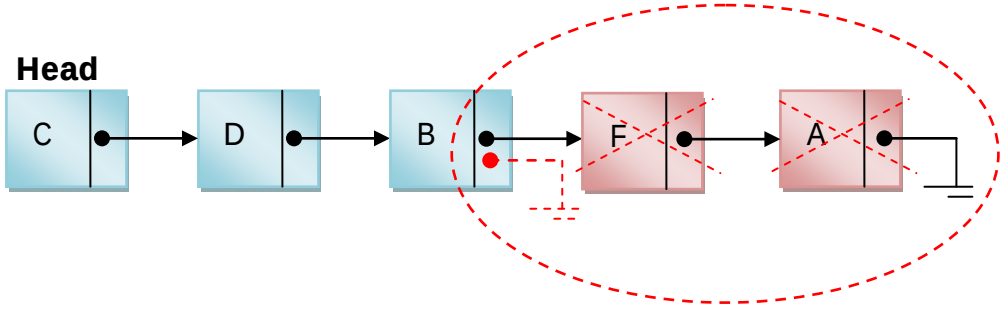
Caso 1: Los nodos a eliminar se encuentran en una posición intermedia dentro de la lista.



Caso 2: Los nodos a eliminar se encuentran al inicio de la lista.



Caso 3: Los nodos a eliminar se encuentran al final de la lista.



Solución en pseudocódigo

```
Si head <> null
    Leer (x,n)
    c ← 1
    p ← head
    Mientras p(liga) <> null
        p ← p(liga)
        c ← c + 1
    Si (x+n) <= c entonces
        p← head
        Si  x = 1 entonces
            j ← 1
            Mientras j <> n
                head ← head(liga)
                eliminar(p)
                p ← head
                j ← j + 1
        de lo contrario
            i← 2
            Mientras i <> x entonces
                i ← i + 1
                p ← p(liga)
            j ← 1
            Mientras j <> n
                q ← p(liga)
                p(liga) ← q(liga)
                eliminar(q)
                j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores')
De lo contrario
    Mensaje (Lista vacía)
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay más de un elemento

Código en C++

```
void lista_sencilla_lineal::eliminar_subcadena(int n, int x)
{
    nodo p,q;
    int c,i,j;
    if (head != NULL)
    {
        c = 1;
        p = head;
        while (p->liga != NULL)
        {
            p = p->liga;
            c++;
        }
        if ((x+n-1)<=c)
        {
            p = head;
            if (x==1)
            {
                j = 0;
                while (j!= n)
                {
                    head = head->liga;
                    delete(p);
                    p=head;
                    j++;
                }
            }
            else
            {
                i = 2;
                while (i!=x)
                {
                    i++;
                    p = p->liga;
                }
                j = 0;
                while (j!=n)
                {
                    q = p->liga;
```

```
                        p->liga = q->liga;
                        delete(q);
                        j++;
                    }
                }
            }
            else
                cout << "Error en los valores";
        }
        else
            cout << "Lista vacía";
}
```

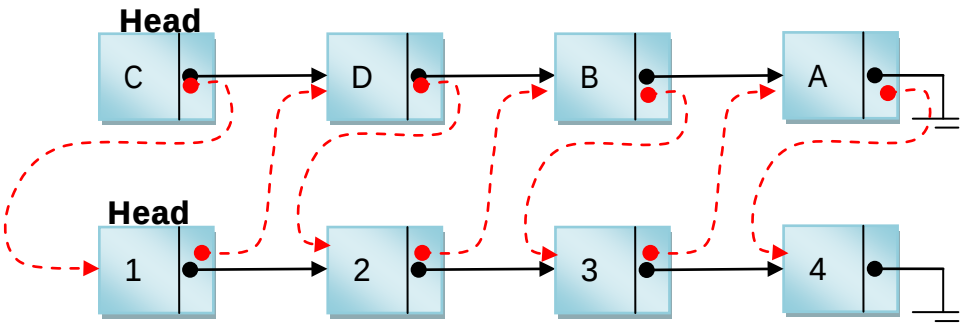
2.16 Intercalar dos listas

Diseñar un algoritmo que permita intercalar dos listas lineales sencillas, para obtener una sola lista lineal sencilla.

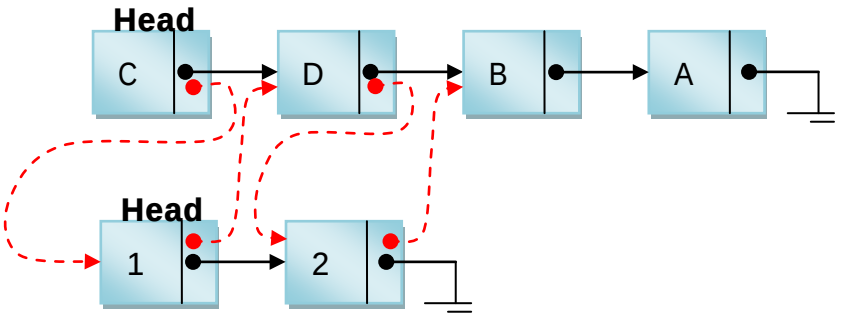
Análisis del problema

La operación de intercalar dos listas lineales se lleva a cabo recorriendo ambas listas al mismo tiempo para ir ligando los elementos de una lista con la otra. Es necesario verificar si las dos listas contienen elementos suficientes para realizar la operación. La solución debe contemplar que las listas no necesariamente tendrán el mismo tamaño.

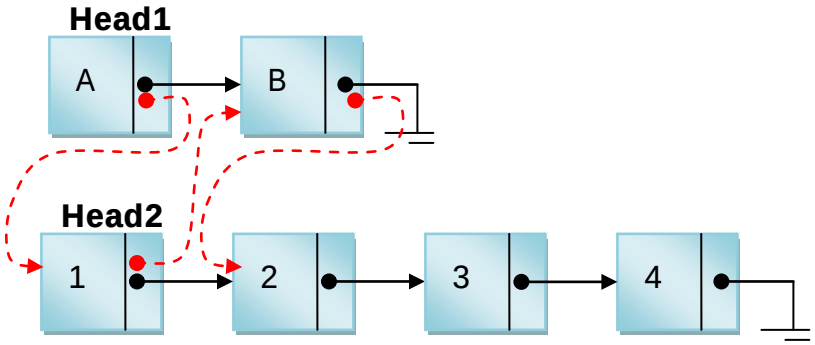
Caso 1: Las dos listas tienen el mismo tamaño.



Caso 2: El tamaño de la primera lista es mayor al tamaño de la segunda lista.



Caso 3: El tamaño de la segunda lista es mayor al tamaño de la primera lista.



Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
    p ← head1
    q ← head2
    Mientras (p(liga) <> null)  y (q(liga) <> null)
        r ← q(liga)
        q(liga) ← p(liga)
        p(liga) ← q
        p ← q(liga)
        q ← r
    Si p(liga) = null entonces
        p(liga) ← q
    De lo contrario
        Si p(liga) <> null y q(liga) = null entonces
            q(liga) ← p(liga)
            p(liga) ← q
    head2 ← null
De lo contrario
    Mensaje ('No hay listas...')
```

Código en C++

```
void lista_sencilla_lineal::intercalar(
lista_sencilla_lineal &a)
{
    nodo p,q,r;
    if (head != NULL && a.head != NULL)
    {
        p = head;
        q = a.head;
        while (p->liga != NULL && q->liga != NULL)
        {
            r = q->liga;
            q->liga = p->liga;
            p->liga = q;
            p = q->liga;
            q = r;
        }
        if (p->liga == NULL)
        {
            p->liga = q;
        }
        else
        if (p->liga != NULL && q->liga == NULL)
        {
            q->liga = p->liga;
            p->liga = q;
        }
        a.head=NULL;
    }
    else
        cout << "No hay listas...";
}
```

2.17 Particionar una lista

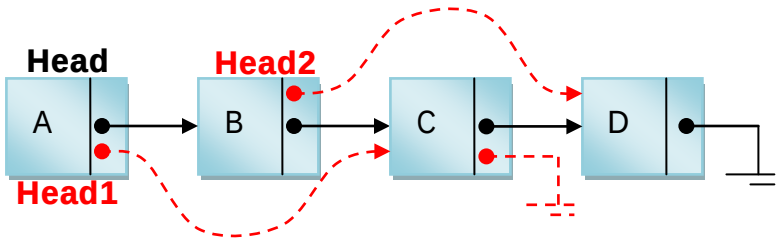
Diseñar un algoritmo que permita particionar en dos listas los nodos de una lista lineal sencilla.

Análisis del problema

La operación de particionar una lista consiste en que tomando como base una lista lineal sencilla se construyen dos listas lineales sencillas pasando los elementos que se encuentren en una posición impar a la primera lista y los elementos que se encuentren en una posición par se pasan a la segunda lista. La solución de este problema se puede hacer de dos maneras:

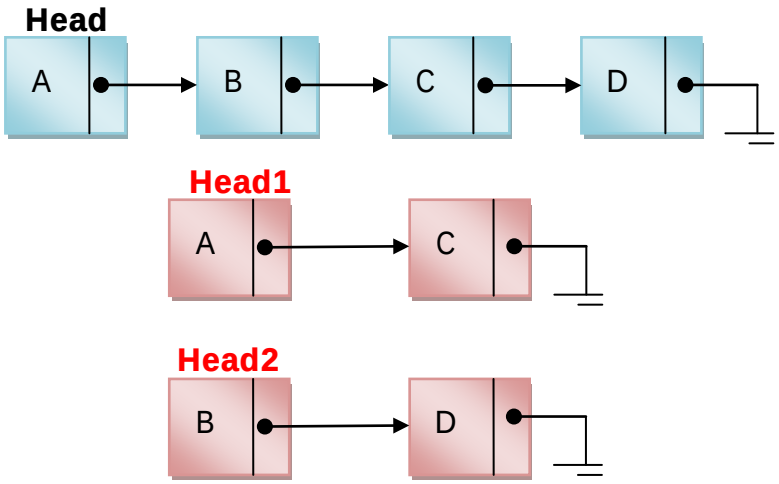
Solución 1:

En esta solución no es necesario crear y eliminar nodos, lo único que se hace es ligar todos los nodos que se encuentran en una posición impar para formar la primera de las listas y todos los nodos que se encuentran en una posición par para formar la segunda de las listas.



Solución 2:

Crear cada uno de los nodos de las dos nuevas listas y copiar los datos que se encuentren en la lista original a la lista que le corresponda, y al final eliminar todos los nodos de la lista original.



Solución en pseudocódigo

Solución 1

```
Si head <> null  y head(liga) <> null entonces
    head1 ← head
    head2 ← head(liga)
    p ← head1
    q ← head2
    Mientras q <> null
        p(liga) ← q(liga)
        p ← q
        q ← q(liga)
De lo contrario
    Mensaje ('No hay suficientes nodos para particionar...')
```

Solución 2

```
Si head <> null  y head(liga) <> null entonces
    p ← head
    q ← head(liga)
    Mientras p <> null
        Nuevo (r)
        r(dato) ← p(dato)
        Si head1 = null entonces
            head1 ← r
            rr ← head1
        de lo contrario
            rr(liga) ← r
            rr ← r
        rr(liga) ← null
        Si q<> null entonces
            Nuevo (s)
            s(dato) ← q(dato)
            Si head2 = null entonces
                head2 ← s
                ss ← head2
            de lo contrario
                s(liga) ← s
                ss ← s
            ss(liga) ← null
        Si q<> null entonces
            p ← q(liga)
            Si p <> null entonces
                q ← q(liga)
            De lo contrario
                q ← null
        De lo contrario
            p ← null
    p ← head
    Mientras p <> null
        head ← head(liga)
        eliminar (p)
        p ← head
De lo contrario
    Mensaje (No hay suficientes nodos para particionar)
```

Código en C++

```
void lista_sencilla_lineal::particionar(
lista_sencilla_lineal &a, lista_sencilla_lineal &b)
{
   nodo p,q,r,rr,s,ss;
   if (head != NULL && head->liga != NULL)
   {
       p = head;
       q = head->liga;
       while (p != NULL)
       {
            r = nuevo();
            r->dato = p->dato;
            if (a.head == NULL)
            {
                a.head = r;
                rr = a.head;
            }
            else
            {
                rr->liga = r;
                rr = r;
            }
            rr->liga = NULL;
            if (q!=NULL)
            {
                s = nuevo();
                s->dato = q->dato;
                if (b.head == NULL)
                {
                    b.head = s;
                    ss = b.head;
                }
                else
                {
                    ss->liga = s;
                    ss = s;
                }
                ss->liga = NULL;
            }
            if (q!=NULL)
            {
```

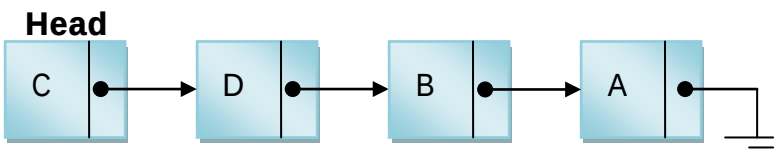
```
                    p = q->liga;
                    if (p != NULL)
                        q = p->liga;
                    else
                        q = NULL;
                }
                else
                    p = NULL;
            }

        p = head;
        while (p != NULL)
            {
                head = head->liga;
                delete (p);
                p = head;
            }
        }
    else
        cout << "No hay suficientes nodos...";
}
```

2.18 Buscar un elemento

Diseñar un algoritmo que permita buscar un elemento x en una lista lineal sencilla.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x, hasta que se encuentre el nodo con el valor de x o que se acabe la lista.



Solución en pseudocódigo

```
Si head <> null
    existe ← falso
    Leer (valor)
    p ← head
    Mientras  p<>null y existe = falso
        Si p(dato) = valor
            existe ← verdadero
        p ← p(liga)
    Si existe = verdadero entonces
        Mensaje (Si se encuentra el elemento x)
    De lo contrario
        Mensaje (No se encuentra el elemento x)
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

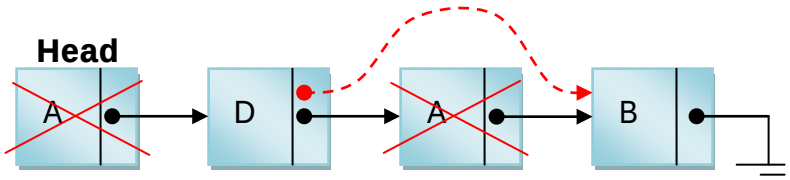
```
int lista_sencilla_lineal::buscar(char valor)
{
   nodo p;
   int existe;
   if (head != NULL)
   {
       existe = false;
       p=head;
       while (p!=NULL && existe == false)
       {
           if (p->dato == valor)
               existe = true;
           p = p->liga;
       }
   }
   else
       cout << "Lista vacía...";
   return (existe);
}
```

2.19 Eliminar repeticiones

Diseñar un algoritmo que permita eliminar todas las repeticiones del elemento x en una lista lineal sencilla.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x. Si al comparar los elementos existe una coincidencia entonces se elimina el nodo, y es necesario ligar el nodo antecesor con el nodo sucesor. Se sugiere que en el ciclo del recorrido antes de que se avance al siguiente nodo, se deje otro apuntador en el nodo, para facilitar el ligado del nodo que se elimina.



Solución en pseudocódigo

```
Si head <> null  entonces
    Leer (x)
    p ← head
    Mientras  p<>null
        Si p(dato) = x entonces
            Si p = head entonces
                head ← head(liga)
                eliminar(p)
                P ← head
            De lo contrario
                q(liga) ← p(liga)
                eliminar(p)
                p ← q(liga)
        De lo contrario
            q ← p
            p ← p(liga)
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

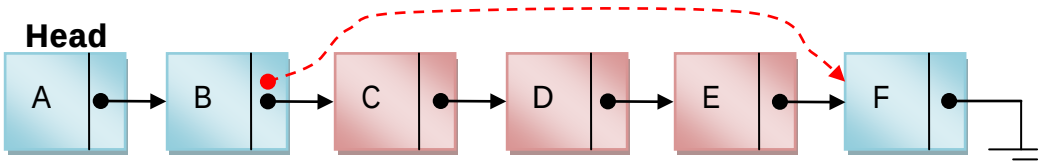
```
void lista_sencilla_lineal::eliminar_repetidos(char valor)
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p!= NULL)
        {
            if (p->dato == valor)
            {
                if (p==head)
                {
                    head = head->liga;
                    delete(p);
                    p = head;
                }
                else
                {
                    q->liga = p->liga;
                    delete(p);
                    p = q->liga;
                }
            }
            else
            {
                q = p;
                p = p->liga;
            }
        }
    }
    else
        cout << "Lista vacía...";
}
```

2.20 Eliminar una subcadena

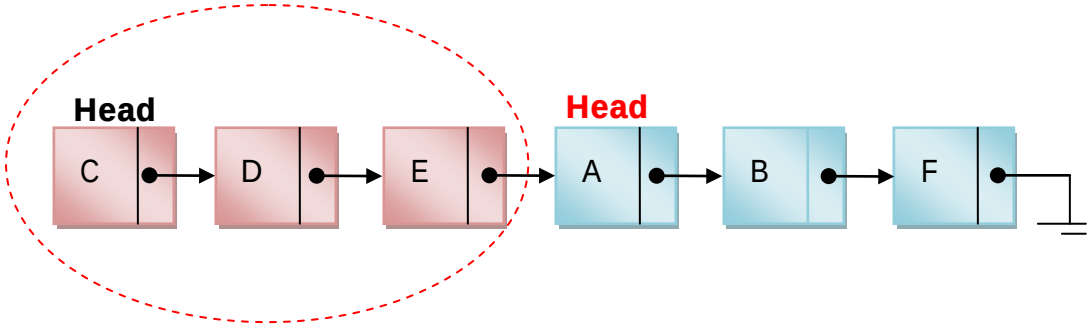
Diseñar un algoritmo que permita eliminar los nodos que formen parte de una subcadena en una lista lineal sencilla.

Análisis del problema

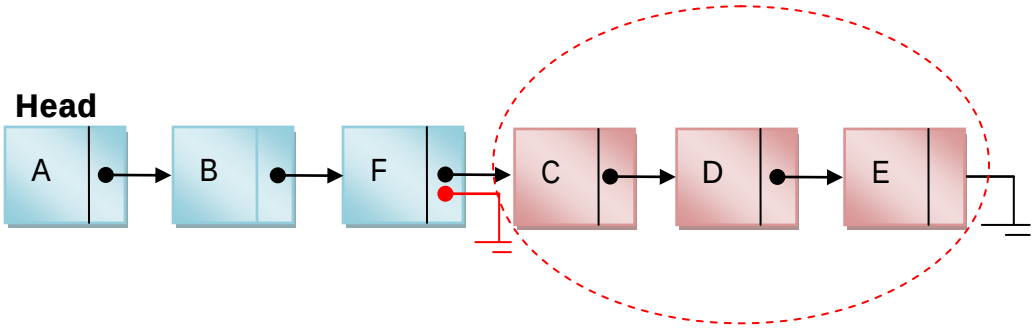
Para resolver este problema es necesario contar con el valor de la subcadena que se va a eliminar y calcular su tamaño. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación. Se localiza el inicio de la subcadena y se eliminan los nodos que forman parte de la subcadena. Es necesario validar si la subcadena está al inicio de la lista, porque en ese caso, es necesario mover la variable head al final de la subcadena.



Subcadena en una posición intermedia



Subcadena al inicio de la lista



Subcadena al final de la lista

Solución en pseudocódigo

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay más

```
Si head <> null
    Leer (subcadena)
    c ← 1, p ← head, tam ← tamaño(subcadena)
    Mientras p(liga) <> null
        p ← p(liga)
        c ← c + 1
    Si c > tam entonces
        p← head, borrado ← falso
        Mientras (p <> null) y (borrado = falso)
            Si  p(dato)  = subcadena[1] entonces
                q ← p, i←0, igual ← verdadero
                Mientras i < tam-1 y igual = verdadero y q<> null
                    q ← q(liga)
                    i ← i+1
                    Si q <> null entonces
                        Si q(dato) <> subcadena[i] entonces
                            igual ← falso
                    de lo contrario
                        igual ← falso
                Si igual = verdadero entonces
                    Si p = head entonces
                        head ← q(liga)
                        Mientras p<> head
                            q ← p(liga)
                            eliminar(p)
                            p ← q
                    de lo contrario
                        r ← head
                        Mientras r(liga) <> p
                            r ← r(liga)
                        r(liga) ← q(liga)
                        Mientras p<>q
                            r←p(liga)
                            eliminar(p)
                            p←r
                        eliminar(p)
                    borrado ← verdadero
            p← p(liga)
        Si borrado = falso entonces
            Mensaje ('Subcadena no existe...')
    De lo contrario
        Mensaje ('Subcadena mayor que la lista...')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_sencilla_lineal::eliminar_subcadena(char *valor)
{
    nodo p,q,r;
    int i,c,tam,borrado,igual;
    c = 1;
    p = head;
    tam = strlen(valor);
    if (head != NULL)
    {
        while (p->liga != NULL)
        {
            p = p->liga;
            c++;
        }
        if (c > tam)
        {
            p = head;
            borrado = false;
            while (p!= NULL && borrado == false)
            {
                if (p->dato == valor[0])
                {
                    q = p;
                    i = 0;
                    igual = true;
                    while (i < tam-1 && igual && q!= NULL)
                    {
                        q = q->liga;
                        i++;
                        if (q != NULL)
                        {
                            if (q->dato != valor[i])
                                igual = false;
                        }
                        else
                            igual = false;
                    }
                if (igual)
                    {
                        if (p== head)
                        {
```

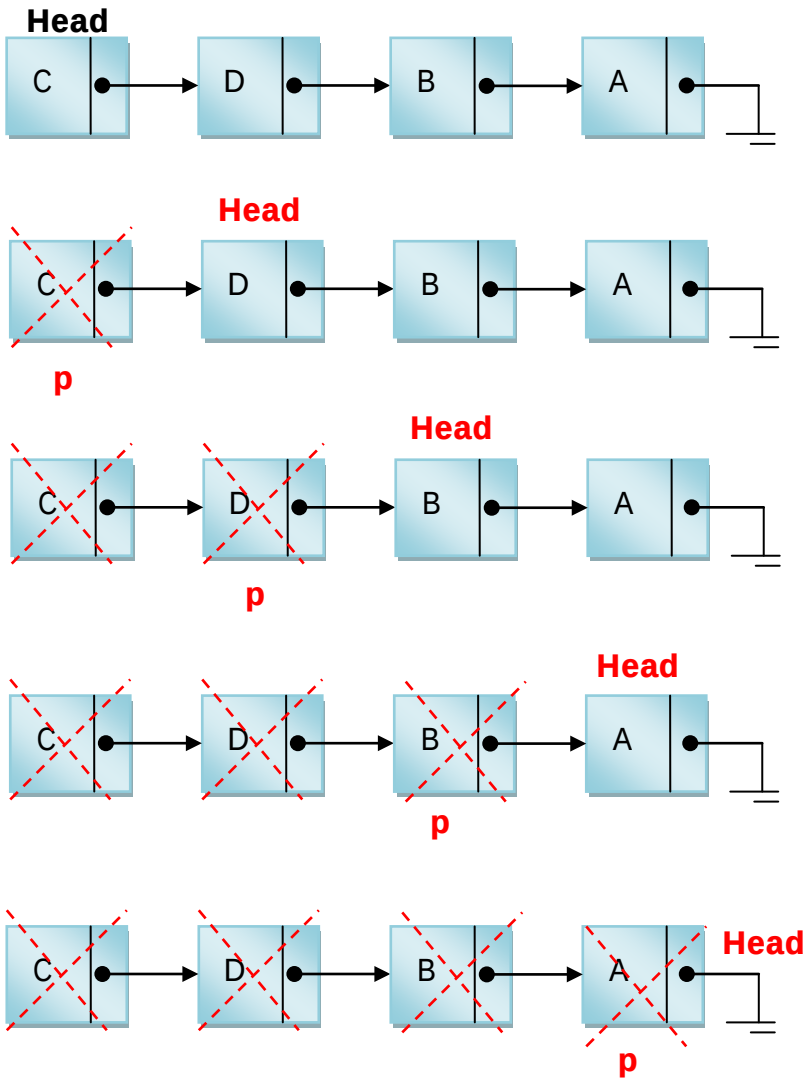
```
                                head = q->liga;
                                while (p!=head)
                                {
                                    q = p->liga;
                                    delete(p);
                                    p=q;
                                }
                            }
                            else
                            {
                                r = head;
                                while (r->liga != p)
                                    r = r->liga;
                                r->liga = q->liga;
                                while (p!=q)
                                {
                                    r=p->liga;
                                    delete(p);
                                    p=r;
                                }
                                delete(p);
                            }
                            borrado=true;
                        }
                    }
                    p=p->liga;
                }
                if (borrado == false)
                    cout << "Subcadena no existe...";
            }
            else
                cout << "Subcadena mayor que la lista...";
        }
        else
            cout << "Lista vacía";
}
```

2.21 Borrar una lista

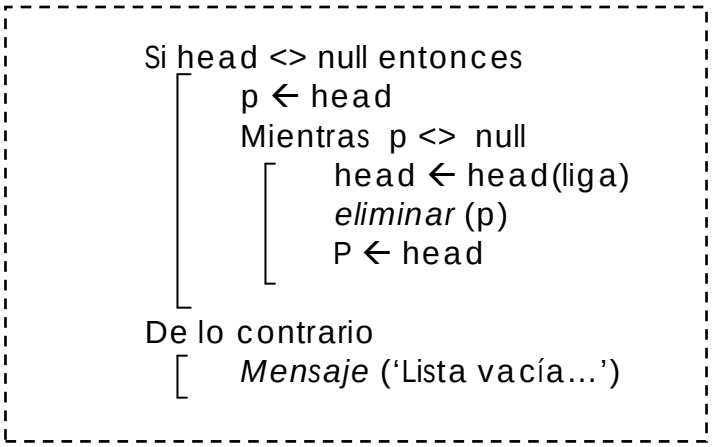
Diseñar un algoritmo que permita borrar todos los elementos de una lista lineal sencilla.

Análisis del problema

Para eliminar todos los nodos de la lista se recorre toda la lista desde el primer nodo hasta llegar al valor nulo, y antes de avanzar hacia el siguiente nodo el *head* se mueve al nodo sucesor y se elimina el nodo. Cuando se eliminan todos los nodos de una lista es necesario inicializar el valor de head en nulo.



Solución en pseudocódigo



Código en C++

```
void lista_sencilla_lineal::eliminar()
{
    nodo p;
    if (head != NULL)
    {
        p = head;
        while (p != NULL)
        {
            head = head->liga;
            delete(p);
            p=head;
        }
    }
    else
        cout << "Lista vacía...";
}
```

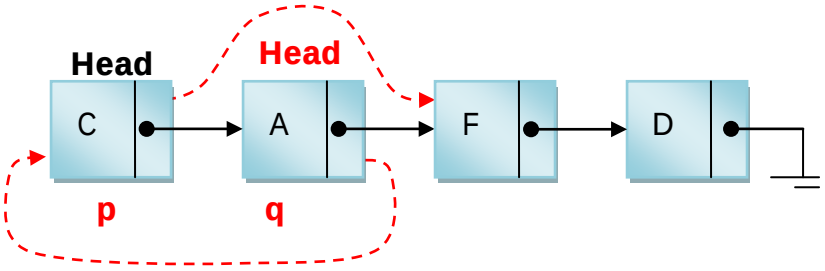
2.22 Ordenar burbuja intercambiando ligas

Diseñar un algoritmo que permita ordenar una lista sencilla lineal con el método de la burbuja, intercambiando las ligas.

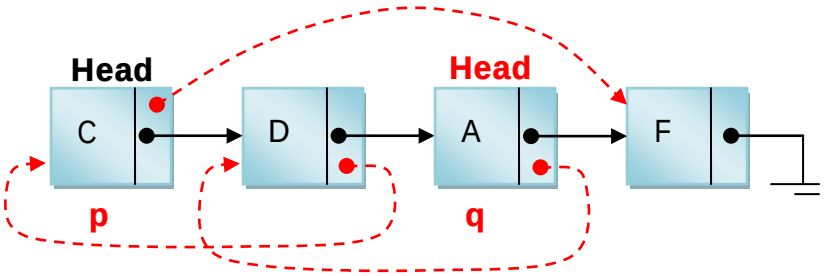
Análisis del problema

Para resolver este problema es necesario contar con al menos dos nodos. Al intercambiar las ligas los valores no cambiaran de localidad de memoria. El método de la burbuja funciona comparando el primer elemento de la lista con el resto de los elementos, en caso de que sea necesario se realiza el intercambio, posteriormente se avanza hacia el segundo elemento para compararlo con el resto y así sucesivamente hasta terminar de comparar todos los elementos. Es necesario considerar cuatro casos posibles cuando se realice un intercambio:

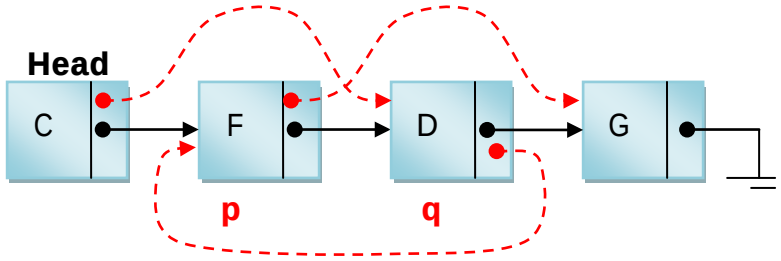
Caso 1: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.



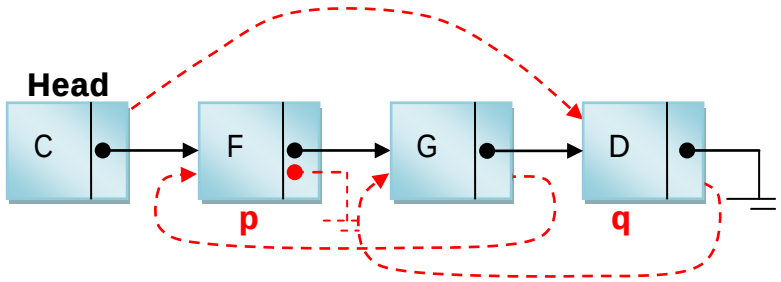
Caso 2: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Caso 3: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.



Caso 4: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras  p(liga) <>  null
        q ← p(liga)
        Mientras q <> null
            Si p(dato) > q(dato) entonces
                Si p = head y p(liga) = q entonces
                    p(liga) ← q(liga)
                    q(liga) ← p
                    head ← q
                    p ← q
                    q ← q(liga)
                Si p = head y p(liga) <> q entonces
                    r ← p(liga)
                    Mientras r(liga) <> q
                        r ← r(liga)
                    r(liga) ← p
                    r ← p(liga)
                    p(liga) ← q(liga)
                    q(liga) ← r
                    head ← q
                    q ← p
                    p ← head
                Si p <> head y p(liga) = q entonces
                    r ← head
                    Mientras r(liga) <> p
                        r ← r(liga)
                    r(liga) ← q
                    p(liga) ← q(liga)
                    q(liga) ← p
                    p ← q
                    q ← q(liga)
                Si p <> head y p(liga) <> q entonces
                    r ← head
                    Mientras r(liga) <> p
                        r ← r(liga)
                    s ← p(liga)
                    Mientras s(liga) <> q
                        s ← s(liga)
                    s(liga) ← p
                    s ← p(liga)
                    p(liga) ← q(liga)
                    q(liga) ← s
                    r(liga) ← q
                    p ← q
                    q ← q(liga)
            q ← q(liga)
        p ← p(liga)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_sencilla_lineal::burbuja_ligas()
{
    nodo p,q,r,s;
    If (head != NULL)
    {
      p = head;
      while (p->liga != NULL)
      {
          q = p->liga;
          while (q != NULL)
          {
              if (p->dato > q->dato)
              {
                  if (p==head && p->liga==q)
                  {
                      p->liga = q->liga;
                      q->liga = p;
                      head = q;
                      p = q;
                      q = q->liga;
                  }
                  else
                  {
                    if (p==head && p->liga != q)
                    {
                      r = p->liga;
                      while (r->liga != q)
                            r = r->liga;
                      r->liga = p;
                      r = p->liga;
                      p->liga = q->liga;
                      q->liga = r;
                      head = q;
                      q = p;
                      p = head;
                    }
                    else
                    {
                      if (p!=head && p->liga == q)
```

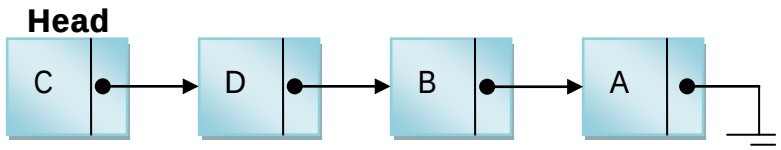
```
                        {
                          r = head;
                          while (r->liga != p)
                             r = r->liga;
                          r->liga = q;
                          p->liga = q->liga;
                          q->liga = p;
                          p = q;
                          q = q->liga;
                        }
                      else
                        {
                          if (p!=head && p->liga != q)
                           {
                             r = head;
                             while (r->liga != p)
                                 r = r->liga;
                             s = p->liga;
                             while (s->liga != q)
                                 s = s->liga;
                             s->liga = p;
                             s = p->liga;
                             p->liga = q->liga;
                             q->liga = s;
                             r->liga = q;
                             p = q;
                             q = q->liga;
                           }
                        }
                    }
                }
              }
              q = q->liga;
           }
           p = p->liga;
       }
    }
    else
        cout << "Lista vacía...";
}
```

2.23 Buscar posición

Diseñar un algoritmo que permite devolver en qué posición se encuentra un carácter en una lista lineal sencilla.

Análisis del problema

Para resolver este problema es necesario contar con el valor del carácter que se buscará en la lista, recorrer toda la lista comparando el valor de cada nodo con el valor del carácter hasta encontrar el final de la lista o hasta que se encuentre el carácter.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (valor)
    p ← head
    pos ← 1
    Mientras (( p(liga) <>  null) y (p(dato)<>valor))
        p ← p(liga)
        pos ← pos + 1
    Si p(dato) = valor entonces
        Mensaje ('Posición = ', pos)
    De lo contrario
        Mensaje ('El valor no se encuentra...')
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

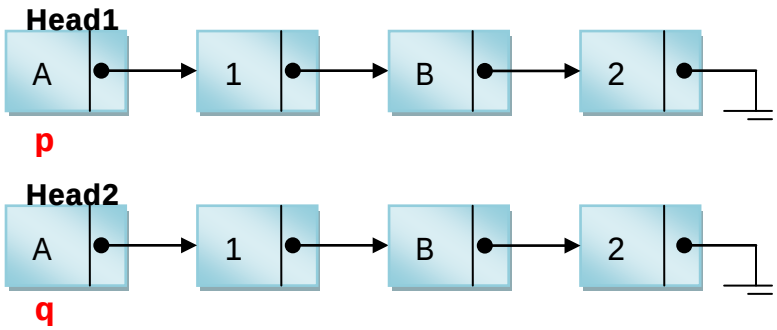
```
int lista_sencilla_lineal::posicion(char valor)
{
   nodo p;
   int pos;
   if (head != NULL)
   {
       p = head;
       pos = 1;
       while (p->liga != NULL && p->dato != valor)
       {
           p = p->liga;
           pos++;
       }
       if (p->dato != valor)
          pos = 0;
    }
    return pos;
}
```

2.24 Comparar dos listas

Diseñar un algoritmo que permita comprobar si dos listas lineales sencillas son exactamente iguales.

Análisis del problema

Para resolver este problema es necesario recorrer ambas listas al mismo tiempo para ir comparando nodo por nodo su valor. Antes de iniciar el recorrido se puede verificar si la longitud de ambas listas es la misma, en caso de que no sea la misma longitud se puede asumir que las listas son diferentes.



Solución en pseudocódigo

```
Si head1 <> null y head2 <> null entonces
    p ← head1
    q ← head2
    igual ← verdadero
    Mientras igual = verdadero y p <>  null y  q<> null
        Si p(dato) <> q(dato) entonces
            igual ← falso
        p ← p(liga)
        q ← q(liga)
    Si igual = verdadero entonces
        Mensaje ('Listas iguales...')
    De lo contrario
        Mensaje ('Las listas no son iguales...')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
int lista_sencilla_lineal::comparar(
lista_sencilla_lineal a)
{
   nodo p,q;
   int igual;
   if (head != NULL && a.head != NULL)
   {
     if (tamano() == a.tamano())
      {
       p = head;
       q = a.head;
       igual = true;
       while (igual && (p!= NULL && q!=NULL))
       {
           if (p->dato != q->dato)
               igual = false;
           p = p->liga;
           q = q->liga;
```

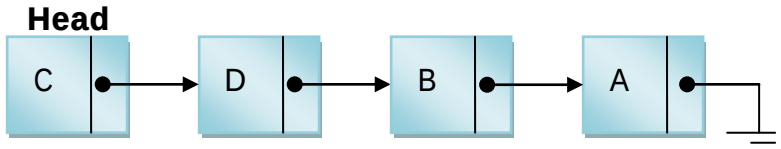
```
        }
      }
     else
        igual = false;
   }
   else
       cout << "Listas vacías...";
   return igual;
}
```

2.25 Reemplazar texto

Diseñar un algoritmo que permita reemplazar parte de una lista sencilla lineal por otro texto.

Análisis del problema

Para resolver este problema es necesario calcular el tamaño de la lista y el tamaño del texto que se va a reemplazar, para determinar si es posible que a partir de la posición que se indique se pueda llevar a cabo la operación. En caso de que los valores sean correctos, se ubica un apuntador en la posición en donde se hará el reemplazo y se inicia con el reemplazo.



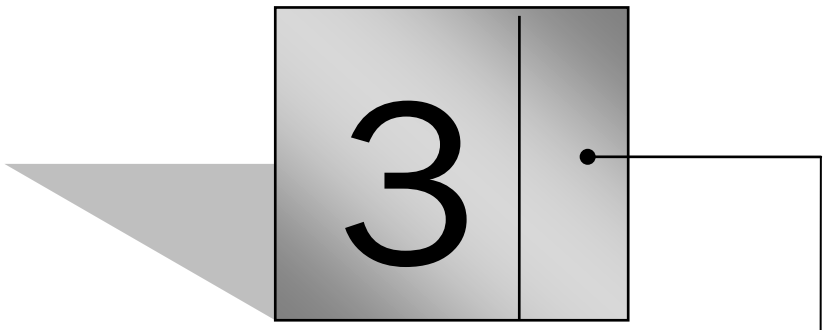
Solución en pseudocódigo

```
Si head <> null
    Leer (pos)
    Leer (texto)
    n ← tamaño(texto)
    c ← 1
    p ← head
    Mientras p(liga) <> null
        p ← p(liga)
        c ← c + 1
    Si (pos+n) <= c entonces
        p← head
        i = 1
        Mientras i <> pos entonces
            i ← i + 1
            p ← p(liga)
        j ← 1
        Mientras j <> n
            p(dato) ← texto[j]
            p ← p(liga)
            j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores')
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

```
void lista_sencilla_lineal::reemplazar(int pos, char *valor)
{
   nodo p;
   int n,c,i,j;
   If (head != NULL)
   {
       n = strlen(valor);
       c = 1;
       p = head;
       while (p->liga != NULL)
       {
           p = p->liga;
           c++;
       }
       if ((pos+n) <= c)
       {
           p = head;
           i = 1;
           while (i!= pos)
           {
               i++;
               p = p->liga;
           }
           j = 0;
           while (j != n)
           {
               p->dato = valor[j];
               p = p->liga;
               j++;
           }
       }
       else
           cout << "Error en los valores...";
   }
   else
    cout << "Lista vacía...";
}
```

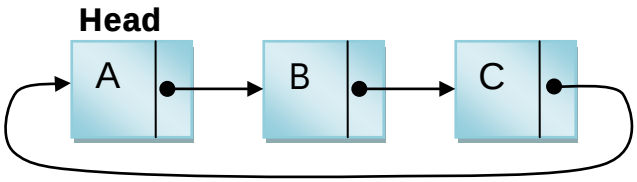
[Capítulo]



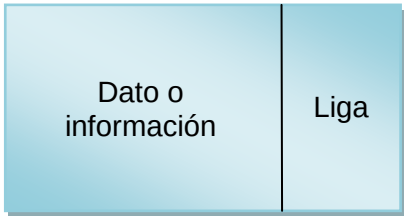
Listas Sencillas Circulares

3 LISTAS SENCILLAS CIRCULARES

La característica principal de una lista sencilla circular es que la liga del último nodo apunta hacia el primer nodo de la lista. El valor nulo solo se utiliza cuando la lista está vacía.



El nodo de una lista sencilla circular debe contener como mínimo dos campos: uno para almacena la información y otro que guarde la dirección de memoria hacia el siguiente nodo de la lista. En la figura se puede apreciar la estructura del nodo para una lista sencilla.



Para definir la estructura del nodo en C++ se hace lo siguiente:

```
struct apuntador
{
  char dato;
  apuntador *liga;
};
```

Para simplificar la asignación de memoria se utiliza la siguiente función:

```
nodo nuevo()
{
 nodo p;
 p = new struct apuntador;
 return p;
}
```

Se presenta la clase lista_sencilla_circular, la cual incluye la variable head y los métodos de las operaciones que se desarrollan en este capítulo para el manejo de las listas sencillas circulares.

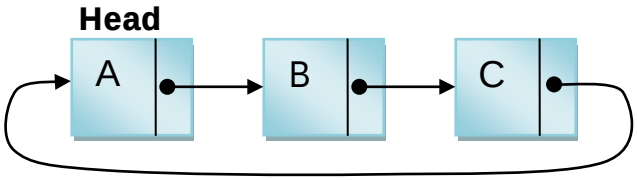
```
class lista_sencilla_circular
{
  nodo head;
  public:
   lista_sencilla_circular();
   void crear();
   void desplegar();
   void mayusculas();
   int tamano();
   void insertar_final();
   void insertar_inicio();
   void insertar(int posicion);
   void borrar_ultimo();
   void borrar_inicio();
   void borrar(int posicion);
   void desplegar_invertida();
   void burbuja();
   void invertir();
   void concatenar(lista_sencilla_circular &b);
   void eliminar_subcadena(int n, int x);
   void intercalar(lista_sencilla_circular &a);
   void particionar(lista_sencilla_circular &a,
        lista_sencilla_circular &b);
   int buscar(char valor);
   void eliminar_repetidos(char valor);
   void eliminar();
   int posicion(char valor);
   int comparar(lista_sencilla_circular a);
   void burbuja_ligas();
   void reemplazar(int pos, char *valor);
   void eliminar_subcadena(char *valor);
};
```

3.1 Crear una lista

Diseñar un algoritmo que permita crear una lista circular sencilla con n número de nodos.

Análisis del problema

Para resolver este problema es necesario la utilización de un ciclo que estará generando cada uno de los nodos que formaran parte de la lista. Es necesario introducir la información de cada uno de los nodos dentro del ciclo. Al final se liga el último nodo con el primer nodo de la lista.



Solución en pseudocódigo

```
Repite
    Nuevo (p)
    Leer (p(dato))
    Si head = null entonces
        head ← p
    De lo contrario
        q(liga) ← p
    p(liga) ← head
    q ← p
    Leer (otro)
Hasta (otro = NO)
```

Código en C++

```
void lista_sencilla_circular::crear()
{
  nodo p,q;
  char otro;
```

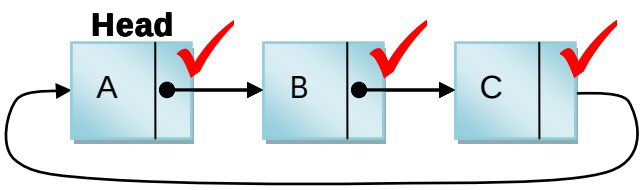
```
  do
  {
     p=nuevo();
     cout << "p(dato) = ";
     cin >> p->dato;
     p->liga = NULL;
     if (head == NULL)
         head = p;
     else
         q->liga = p;
     p->liga = head;
     q=p;
     cout << "Capturar otro nodo s/n ? " ;
     cin >> otro;

  } while (otro == 's');
}
```

3.2 Recorrer una lista

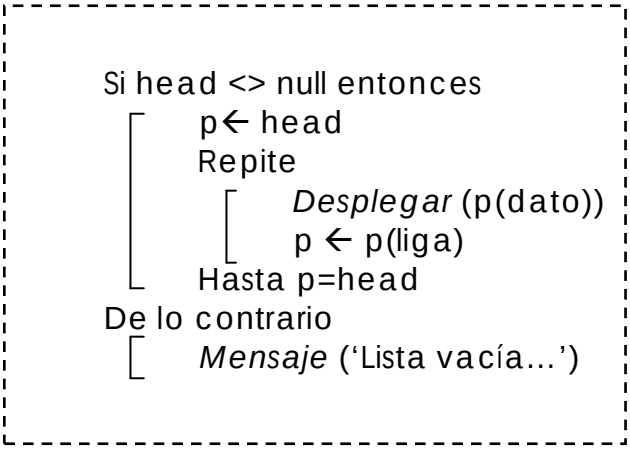
Diseñar un algoritmo que permita desplegar el contenido de todos los nodos una lista circular sencilla.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se recorre toda la lista desde el primer nodo donde se encuentra *head*. En una lista circular sencilla no existe el valor nulo, entonces para encontrar el final de la lista es necesario hacer referencia al primer nodo de la lista y tomar en cuenta esto para la condición del ciclo que recorrerá toda la lista.



Solución en pseudocódigo



Código en C++

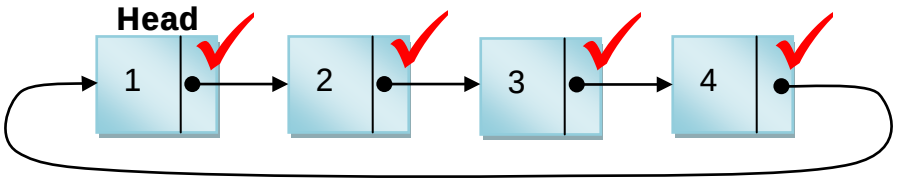
```
void lista_sencilla_circular::desplegar()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    do
    {
       cout << p->dato;
       p = p->liga;
    } while (p!=head);
  }
  else
    cout <<  "Lista Vacía";
}
```

3.3 Calcular tamaño

Diseñar un algoritmo que permita determinar el tamaño de una lista sencilla circular.

Análisis del problema

Para calcular el tamaño de la lista es necesario recorrer todos los nodos de la lista desde el primer nodo hasta encontrar el nodo cuya liga apunta al primer nodo de la lista. Para contar el total de nodos se utiliza un contador que se va incrementando.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    total ← 1
    Mientras p(liga) <> head
        p ← p(liga)
        total ← total + 1
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

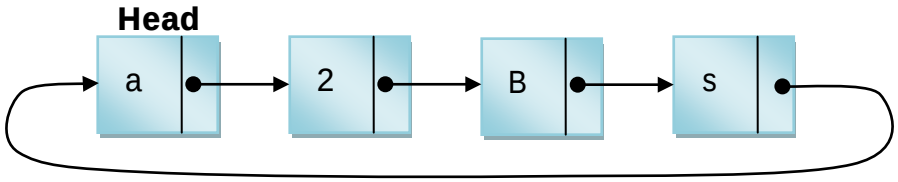
```
int lista_sencilla_circular::tamano()
{
  nodo p;
  int total;
  if (head != NULL)
  {
    p = head;
    total = 1;
    while (p->liga !=head)
    {
       p = p->liga;
       total++;
    }
  }
  else
    cout <<  "Lista Vacía";
  return total;
}
```

3.4 Convertir a mayúsculas

Diseñar un algoritmo que permita convertir todos los elementos alfabéticos de una lista sencilla circular de minúsculas a mayúsculas.

Análisis del problema

Para resolver este problema es necesario recorrer todos los nodos de la lista desde el inicio, para ir comparando el valor del nodo y en caso de que sea una letra convertirla a mayúscula.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Repite
        Si p(dato) es una letra entonces
            p(dato) ← mayúscula(p(dato))
        p ← p(liga)
    Hasta p = head
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

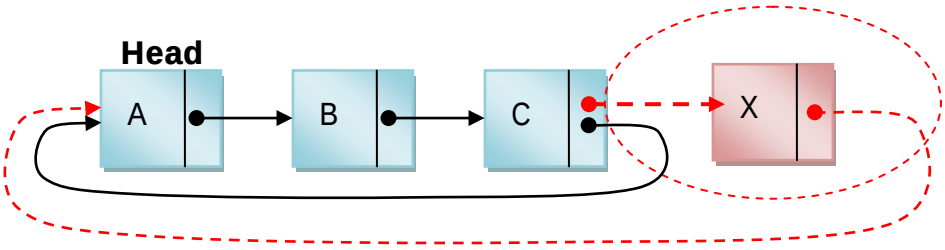
```
void lista_sencilla_circular::mayusculas()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    do
    {
       if (p->dato>='a' && p->dato <= 'z')
            p->dato -=32;
       p = p->liga;
    } while (p!=head);
  }
  else
    cout <<  "Lista Vacía";
}
```

3.5 Insertar al final

Diseñar un algoritmo que permita insertar un nodo al final de una lista circular sencilla.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía es necesario recorrer toda la lista para ubicarse en el último nodo, crear el nuevo nodo, ligar el último nodo con el nuevo nodo y el nuevo nodo con el primer nodo. Si la lista está vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo.



Solución en pseudocódigo

```
Nuevo (p)
Leer (p(dato))
p(liga) ← head
Si head <> null entonces
    q ← head
    Mientras q(liga) <> head
        q ← q(liga)
    q(liga)← p
De lo contrario
    head ← p
```

Código en C++

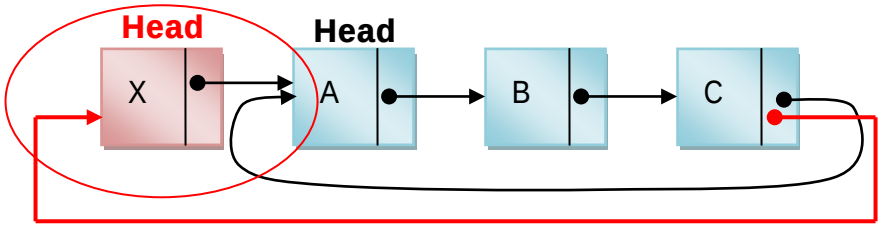
```
void lista_sencilla_circular::insertar_final()
{
   nodo p,q;
   p = nuevo();
   cout << "p(dato) = ";
   cin >> p->dato;
   p->liga = head;
   if (head != NULL)
   {
        q = head;
        while (q->liga != head)
            q = q->liga;
        q->liga = p;
   }
   else
        head = p;
}
```

3.6 Insertar al inicio

Diseñar un algoritmo que permita insertar un nodo al inicio de una lista circular sencilla.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se crea el nuevo nodo y se liga con el primer nodo de la lista y *head* se mueve al nuevo nodo. Es necesario recorrer toda la lista para hacer que el último nodo de la lista apunte al nuevo nodo del inicio. Si la lista está vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo.



Solución en pseudocódigo

```
Nuevo (p)
Leer (p(dato))
Si head = null entonces
    p(liga) ← p
De lo contrario
    q ← head
    Mientras q(liga) <> head
        q ← q(liga)
    p(liga) ← head
    q(liga) ← p
head ← p
```

Código en C++

```
void lista_sencilla_circular::insertar_inicio()
{
    nodo p,q;
    p = nuevo();
    cout << "p(dato) = ";
    cin >> p->dato;
    if (head == NULL)
    {
        p->liga = p;
    }
    else
    {
        q = head;
        while (q->liga != head)
        {
            q = q->liga;
        }
        p->liga = head;
        q->liga = p;
    }
    head = p;
}
```

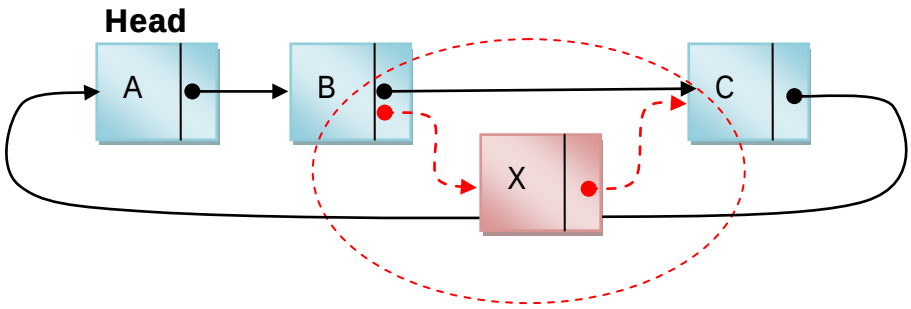
3.7 Insertar en cualquier posición

Diseñar un algoritmo que permita insertar un nodo en cualquier posición en una lista circular sencilla.

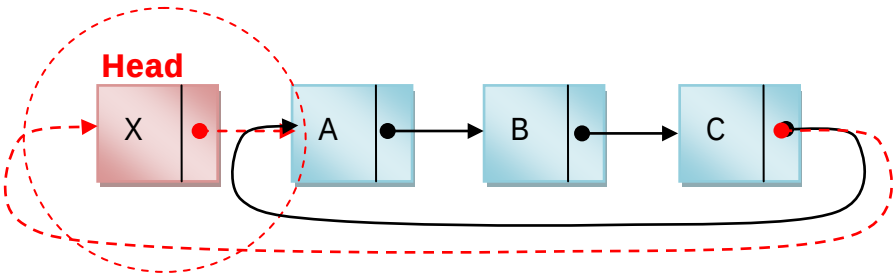
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos y la posición en la que se desea insertar el nuevo nodo es válida, es menor o igual al total de los nodos de la lista. La solución debe contemplar los casos siguientes:

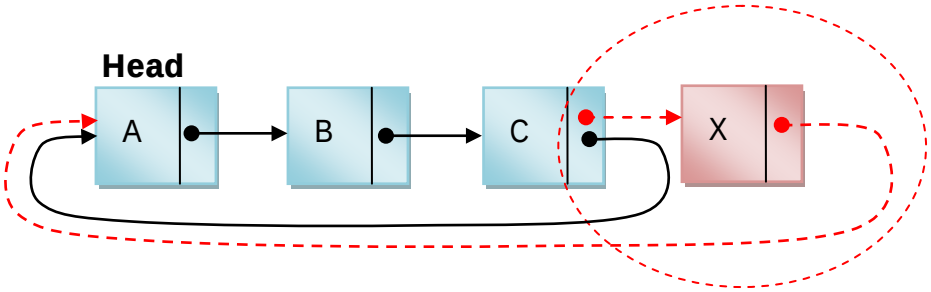
Caso 1: Insertar el nuevo nodo en una posición intermedia dentro de la lista.



Caso 2: Insertar el nuevo nodo al inicio de la lista. En este caso también se debe determinar si el nodo que se inserta es el primero de la lista, y en su caso, ubicar a head en dicho nodo y hacer que la liga del último nodo apunte hacia el nuevo nodo.



Caso 3: Insertar el nuevo nodo al final de la lista y hacer que la liga del nuevo nodo apunte hacia el primer nodo de la lista donde se encuentra head.



Solución en pseudocódigo

```
Leer (posición)
p ← head
c ← 1
Mientras p(liga) <> head
    p ← p(liga)
    c ← c + 1
Si (posición  > 0) y (posición <= c+1) entonces
    Nuevo (p)
    Leer (p(dato))
    Si pos = 1 entonces
        p(liga) ← head
        q ← head
        mientras q(liga) <> head
            q ← q(liga)
        head ← p
        q(liga) ← head
    De lo contrario
        q ← head
        para i = 1 hasta pos – 2
            q ← q(liga)
        p(liga) ← q(liga)
        q(liga) ← p
De lo contrario
    Mensaje ('Posición incorrecta...')
```

Código en C++

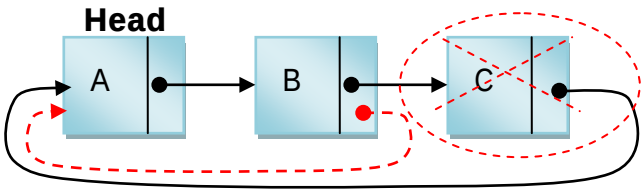
```
void lista_sencilla_circular::insertar(int posicion)
{
   nodo p,q;
   int c,i;
   p = head;
   c = 1;
   while (p->liga != head)
   {
       p = p->liga;
       c++;
   }
   if ((posicion > 0) && (posicion <= c+1))
   {
       p = nuevo();
       cout << "p(dato) = ";
       cin >> p->dato;
       if (posicion==1)
       {
           p->liga = head;
           q = head;
           while (q->liga != head)
           {
               q = q->liga;
           }
           head = p;
           q->liga = head;
       }
       else
       {
           q = head;
           for(i=1; i<=posicion-2; i++)
                q = q->liga;
           p->liga = q->liga;
           q->liga = p;
       }
   }
   else
        cout <<  "Posición Incorrecta...";
}
```

3.8 Borrar el último nodo

Diseñar un algoritmo que permita borrar el último nodo de una lista circular sencilla.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene al menos un elemento. Es necesario recorrer los nodos de la lista para ubicarse en la penúltima posición, eliminar el último nodo y hacer que la liga del penúltimo nodo apunte hacia el primer nodo de la lista. En el caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *Head* en nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras (p(liga) <> head)
        q ← p
        p ← p(liga)
    Si p<>head entonces
        q(liga) ← head
    De lo contrario
        head ← null
    Eliminar (p)
De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

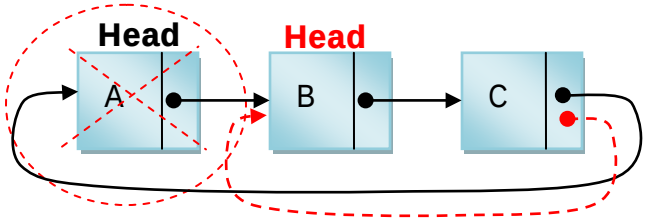
```
void lista_sencilla_circular::borrar_ultimo()
{
  nodo p,q;
  if (head != NULL)
  {
    p = head;
    while (p->liga != head)
    {
        q = p;
        p = p->liga;
    }
    if (p != head)
        q->liga = head;
    else
        head = NULL;
    delete(p);
  }
  else
    cout <<  "Lista Vacía";
}
```

3.9 Borrar el primer nodo

Diseñar un algoritmo que permita borrar el primer nodo de una lista circular sencilla.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene al menos un elemento. Si la lista contiene elementos se posiciona un apuntador en el último nodo de la lista, se elimina el primer nodo de la lista, la variable *head* se mueve al siguiente nodo de la lista y el último nodo de la lista se liga con el primer nodo de la lista. En caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *head* en nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    q ← head
    Mientras q(liga) <> head
        q ← q(liga)
    p ← head
    Si q = head entonces
        Head ← null
    De lo contrario
        head ← head(liga)
        q(liga) ← head
    Eliminar (p)

De lo contrario
    Mensaje ('No hay elementos...')
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay más de un elemento

Código en C++

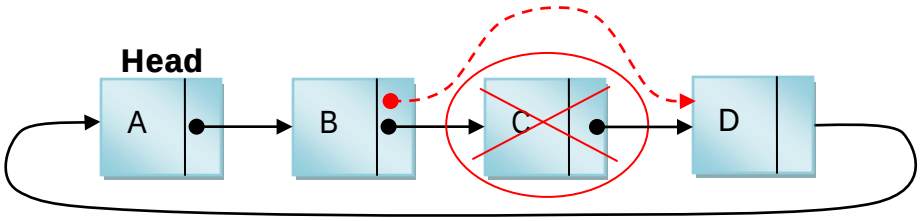
```
void lista_sencilla_circular::borrar_inicio()
{
  nodo p,q;
  if (head != NULL)
  {
    q = head;
    while (q->liga != head)
    {
        q = q->liga;
    }
    p = head;
    if (q==head)
    {
        head = NULL;
    }
    else
    {
        head = head->liga;
        q->liga = head;
    }
    delete(p);
  }
  else
    cout <<  "No hay elementos";
}
```

3.10 Borrar cualquier nodo

Diseñar un algoritmo que permita borrar un nodo en cualquier posición en una lista circular sencilla.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos y la posición del elemento que se desea eliminar es válida. En el caso de que la lista contenga un solo nodo, se elimina el nodo y se inicializa la variable head en nulo. Si la posición es la última o la primera se utilizan los algoritmos para eliminar el último o el primero nodo. Si la posición es intermedia es necesario ligar el nodo antecesor y el sucesor del nodo que se elimina.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (pos)
    p ← head
    c ← 1
    Mientras (c <> pos) y (p(liga) <> head)
        q ← p
        p ← p(liga)
        c ← c + 1
    Si c = pos entonces
        Si p = head entonces
            Si p(liga) <> head entonces
                head ← head (liga)
                q ← head
                Mientras q(liga) <> p
                    q ← q(liga)
                q(liga) ← head
            De lo contrario
                head ← null
        De lo contrario
            q(liga) ← p(liga)
        Eliminar (p)
    De lo contrario
        Mensaje ('Posición inválida...')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_sencilla_circular::borrar(int posicion)
{
 nodo p,q;
 int c;
 if (head != NULL)
 {
   p = head;
```

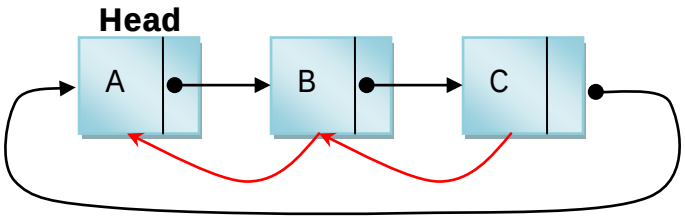
```
   c = 1;
   while ((c != posicion) && (p->liga != head))
   {
       q = p;
       p = p->liga;
       c++;
   }
   if (c==posicion)
   {
       if (p==head)
       {
           if (p->liga != head)
           {
               head = head->liga;
               q = head;
               while (q->liga != p)
                    q = q->liga;
               q->liga = head;
           }
           else
                head = NULL;
       }
       else
            q->liga = p->liga;
       delete (p);
   }
   else
       cout <<  "Posición inválida...";
 }
 else
   cout <<  "Lista Vacía";
}
```

3.11 Desplegar invertida

Diseñar un algoritmo que permita desplegar el contenido de todos los nodos una lista circular sencilla de forma invertida.

Análisis del problema

Para resolver este problema es necesario verificar si la lista contiene elementos. Para desplegar el contenido de la lista en forma inversa se tiene que ir recorriendo la lista de atrás hacia adelante, pero como la lista no está ligada en esa dirección, se utilizan dos apuntadores uno se deja en la última posición y el otro en la penúltima posición. Después el apuntador que se encuentra en la última posición se ubica en el penúltimo nodo y se recorre nuevamente la lista desde el inicio dejando un apuntador un nodo antes del penúltimo, y así sucesivamente hasta llegar al primer nodo de la lista.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(liga) <> head
        p← p(liga)
    Mientras (p <> head)
        q ← head
        Mientras q(liga) <> p
            q ← q(liga)
        Desplegar (p(dato))
        p ← q
    Desplegar (p(dato))

De lo contrario
    Mensaje ('No hay elementos...')
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay mas de un elemento

Código en C++

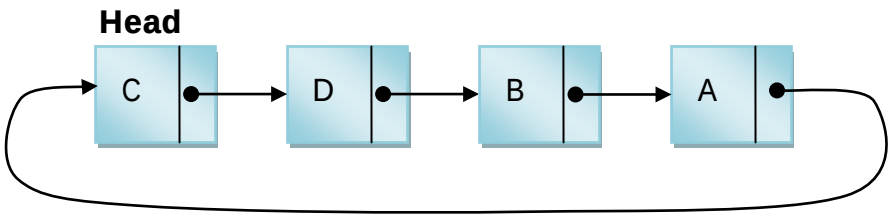
```
void lista_sencilla_circular::desplegar_invertida()
{
   nodo p,q;
   if (head != NULL)
   {
      p = head;
      while (p->liga != head)
        p = p->liga;
      while (p != head)
      {
          q = head;
          while (q->liga != p)
            q = q->liga;
          cout << p->dato;
          p = q;
      }
      cout << p->dato;
   }
   else
      cout <<  "Lista Vacía";
}
```

3.12 Ordenar burbuja

Diseñar un algoritmo que permita ordenar una lista sencilla circular utilizando el método de la burbuja.

Análisis del problema

Para la implementación del método de ordenación de la burbuja se requieren dos ciclos anidados para ir comparando los elementos y hacer los intercambios que sean necesarios.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(liga) <> head
        q ← p(liga)
        Mientras (q <> head)
            Si (q(dato < p(dato) entonces
                aux ← p(dato)
                p(dato) ← q(dato)
                q(dato) ← aux
            q ← q(liga)
        p ← p(liga)
De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

```
void lista_sencilla_circular::burbuja()
{
  nodo p,q;
  char aux;
  if (head != NULL)
  {
      p = head;
      while (p->liga !=head)
      {
          q = p->liga;
          while (q != head)
          {
              if (q->dato < p->dato)
              {
                  aux = p->dato;
                  p->dato = q->dato;
                  q->dato = aux;
              }
              q = q->liga;
          }
          p = p->liga;
      }
  }
  else
      cout <<  "No hay elementos";
}
```

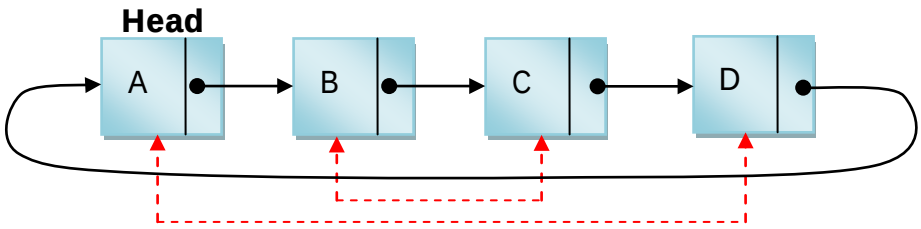
3.13 Invertir la lista

Diseñar un algoritmo que permita invertir los nodos de una lista sencilla circular.

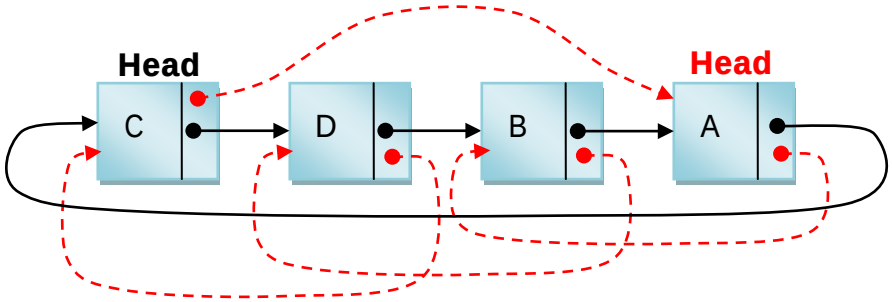
Análisis del problema

La operación de invertir una lista puede hacerse de dos maneras: la primera intercambiando únicamente los datos y la segunda dejando los datos en la posición de memoria en la que se encuentran y cambiar el ligado de los nodos para que la lista quede invertida.

a) Moviendo datos



b) Moviendo ligas



Solución en pseudocódigo

a) Moviendo datos

```
Si head <> null  y head(liga) <> head entonces
    p ← head
    Mientras p(liga) <> head
        p ← p(liga)
    r ← p
    q ← head
    Repite
        aux ← p(dato)
        p(dato) ← q(dato)
        q(dato) ← aux
        s ← q
        Mientras s(liga) <> p
            s ← s(liga)
        p(liga) ← s
        q ← q(liga)
    Hasta (p = q) o (p(liga) = q
    head ← r
De lo contrario
    Mensaje ('No hay suficientes elementos...')
```

b) Moviendo ligas

```
Si head <> null  y head(liga) <> head entonces
    p ← head
    Mientras p(liga) <> head
        p ← p(liga)
    r ← p
    Repite
        q ← head
        Mientras q(liga) <> p
            q ← q(liga)
        p(liga) ← q
        p ← q
    Hasta p = head
    p(liga) ← r
    head ← r
De lo contrario
    Mensaje ('No hay suficientes elementos...')
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay mas de un elemento

Código en C++

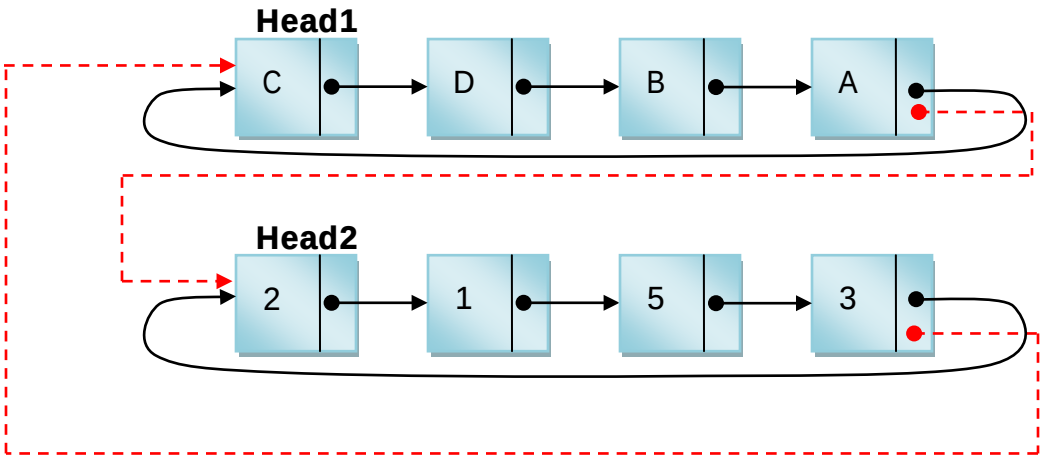
```
void lista_sencilla_circular::invertir()
{
   nodo p,q,r;
   if ((head != NULL) && (head->liga != head))
   {
       p = head;
       while (p->liga != head)
           p = p->liga;
       r = p;
       do
       {
           q = head;
           while (q->liga != p)
                q = q->liga;
           p->liga = q;
           p = q;
       } while (p != head);
       p->liga = r;
       head = r;
   }
   else
       cout <<  "No hay suficientes elementos";
}
```

3.14 Concatenar dos listas

Diseñar un algoritmo que permita concatenar los nodos de dos listas sencillas circulares.

Análisis del problema

Para resolver este problema primero se debe verificar si existen dos listas sencillas circulares con al menos un nodo cada una de ellas. Después es necesario posicionar un apuntador en el último nodo de la primera lista y otro apuntador en el último nodo de la segunda lista. La liga del último nodo de la primera lista se hace que apunte al primer nodo de la segunda lista y la liga del último nodo de la segunda lista se direcciona al primer nodo de la primera lista.



Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
   [   p ← head1
       Mientras p(liga) <> head1
          [ p ← p(liga)
       p(liga) ← head2
       Repite
          [ p ← p(liga)
       Hasta  p(liga) = head2
       p(liga) ← head1
De lo contrario
   [   Mensaje ('No hay listas...')
```

Código en C++

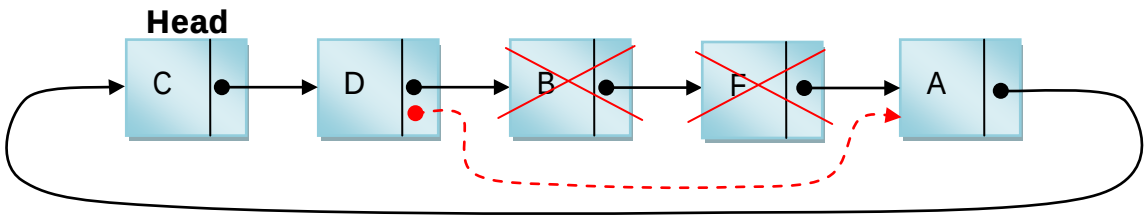
```
void lista_sencilla_circular::concatenar(
lista_sencilla_circular &b)
{
   nodo p;
   if (head != NULL && b.head != NULL)
   {
       p = head;
       while (p->liga != head)
            p = p->liga;
       p->liga = b.head;
       do
       {
           p = p->liga;
       } while (p->liga != b.head);
       p->liga = head;
       b.head = NULL;
   }
   else
       cout <<  "No hay listas";
}
```

3.15 Eliminar n número de nodos

Diseñar un algoritmo que permita hacer la función de eliminar n número de nodos a partir de la posición x en una lista sencilla circular.

Análisis del problema

Para resolver este problema es necesario contar con el valor de n y x. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación, es decir, si existe la posición a partir de la cual se van a eliminar elementos y suficientes nodos para cumplir con el valor de n.



Solución en pseudocódigo

```
Si head <> null
    Leer (x,n)
    c ← 1
    p ← head
    Mientras p(liga) <> head
        p ← p(liga)
        c ← c + 1
    q ← p
    Si (x+n-1) <= c entonces
        p← head
        Si  x = 1 entonces
            q ← head
            Mientras q(liga) <> head
                q ← q(liga)
            j ← 0
            Mientras j <> n
                head ← head(liga)
                Eliminar(p)
                p ← head
                j ← j + 1
            q(liga) ← head
        de lo contrario
            i← 2
            Mientras i <> x entonces
                i ← i + 1
                p ← p(liga)
            j ← 0
            Mientras j <> n
                q ← p(liga)
                p(liga) ← q(liga)
                Eliminar(q)
                j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores...')
De lo contrario
    Mensaje ('Lista vacía...')
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay más de un elemento

Código en C++

```
void lista_sencilla_circular::eliminar_subcadena(int n, int x)
{
    nodo p,q;
    int c,i,j;
    if (head != NULL)
    {
        c = 1;
        p = head;
        while (p->liga != head)
        {
            p = p->liga;
            c++;
        }
        if ((x+n-1)<=c)
        {
            p = head;
            if (x==1)
            {
                q = head;
                while (q->liga != head)
                    q = q->liga;
                j = 0;
                while (j!= n)
                {
                    head = head->liga;
                    delete(p);
                    p=head;
                    j++;
                }
                q->liga = head;
            }
          else
            {
                i = 2;
                while (i!=x)
                {
                    i++;
                    p = p->liga;
                }
```

```
                    j = 0;
                    while (j!=n)
                    {
                        q = p->liga;
                        p->liga = q->liga;
                        delete(q);
                        j++;
                    }
                }
            }
            else
                cout << "Error en los valores";
        }
        else
            cout << "Lista vacía";
}
```

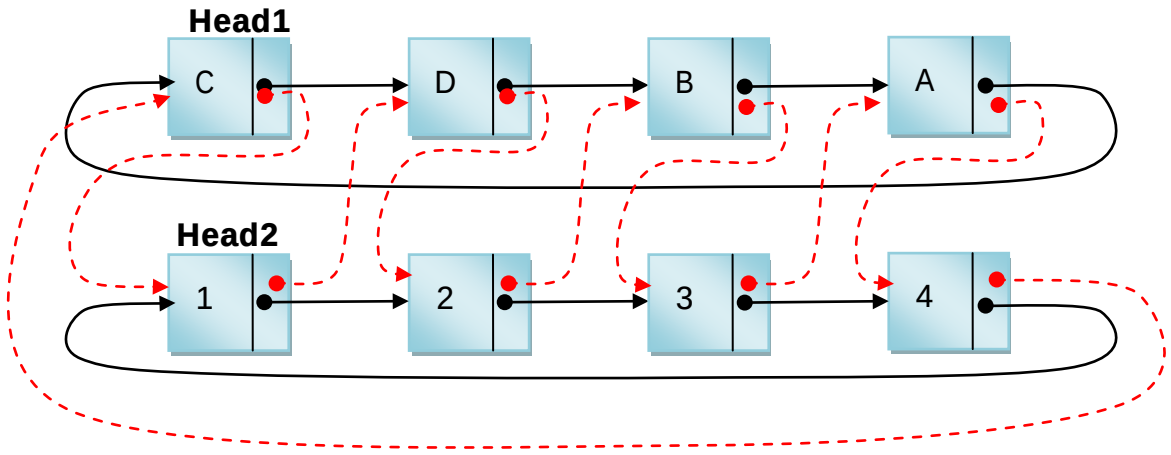
3.16 Intercalar dos listas

Diseñar un algoritmo que permita Intercalar los nodos de dos listas sencillas circulares

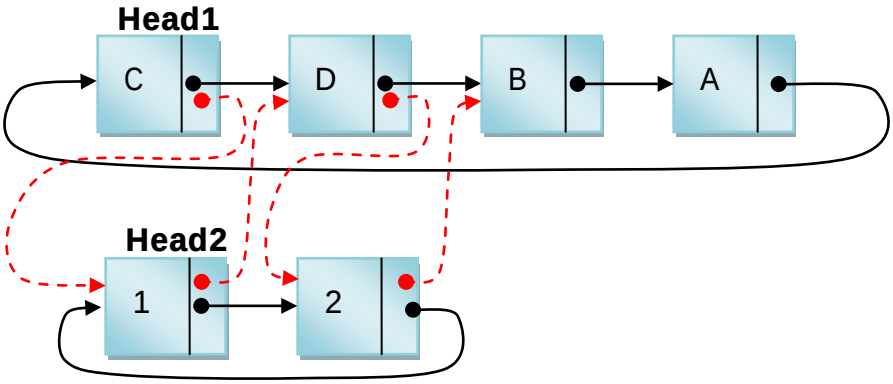
Análisis del problema

La operación de intercalar dos listas circulares se lleva a cabo recorriendo ambas listas al mismo tiempo. Es necesario verificar si las dos listas contienen elementos.

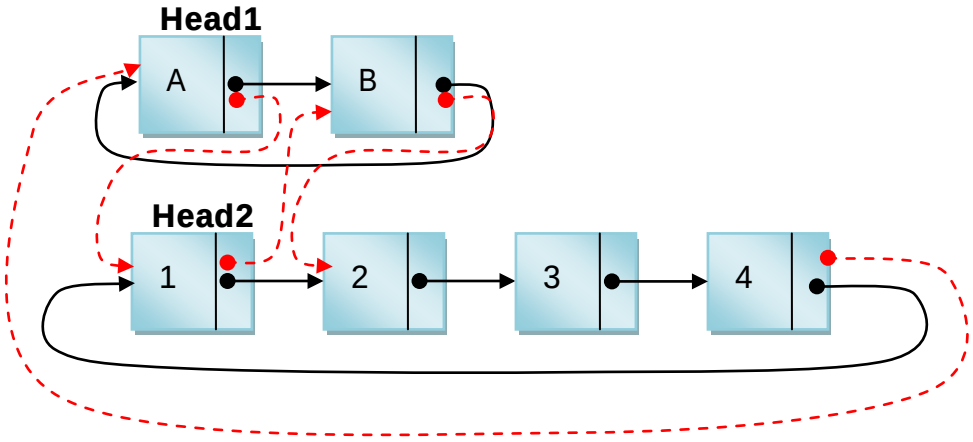
Caso 1: Las dos listas tienen el mismo tamaño.



Caso 2: El tamaño de la primera lista es mayor al tamaño de la segunda lista.



Caso 3: El tamaño de la segunda lista es mayor al tamaño de la primera lista.



Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
    p ← head1
    q ← head2
    Mientras (p(liga) <> head1)  y (q(liga) <> head2)
        r ← q(liga)
        q(liga) ← p(liga)
        p(liga) ← q
        p ← q(liga)
        q ← r
    Si p(liga) = head1 y q(liga) = head2 entonces
        p(liga) ← q
        q(liga) ← head1
    De lo contrario
        Si p(liga) <> head1 y q(liga) = head2 entonces
            q(liga) ← p(liga)
            p(liga) ← q
        De lo contrario
            Si p(liga) = head1 y q(liga) <> head2 entonces
                p(liga) ← q
                Mientras p(liga) <> head2
                    p ← p(liga)
                p(liga) ← head1
De lo contrario
    Mensaje ('No hay listas...')
```

Código en C++

```
void lista_sencilla_circular::intercalar(
lista_sencilla_circular &a)
{
    nodo p,q,r;
    if (head != NULL && a.head != NULL)
    {
        p = head;
```

```
        q = a.head;
        while (p->liga != head && q->liga != a.head)
        {
            r = q->liga;
            q->liga = p->liga;
            p->liga = q;
            p = q->liga;
            q = r;
        }
        if (p->liga == head  && q->liga == a.head)
        {
            p->liga = q;
            q->liga = head;
        }
          else
        {
            if (p->liga != head && q->liga == a.head)
            {
                q->liga = p->liga;
                p->liga = q;
            }
            else
            {
                if ((p->liga == head) && (q->liga != a.head))
                {
                    p->liga = q;
                    while (p->liga != a.head)
                        p = p->liga;
                    p->liga = head;
                }
            }
        a.head=NULL;
        }
    }
    else
        cout << "No hay listas...";
}
```

3.17 Particionar una lista

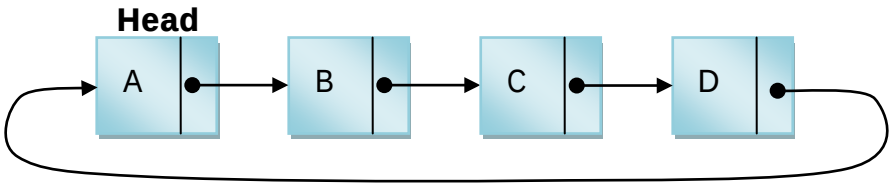
Diseñar un algoritmo que permita particionar los nodos una lista sencilla circular en dos listas sencillas circulares.

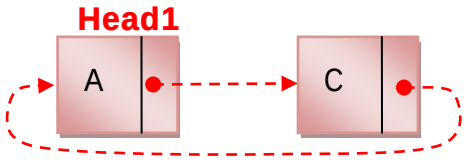
Análisis del problema

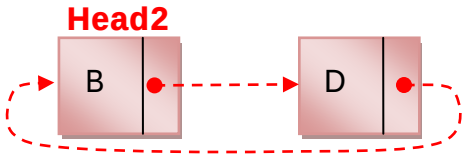
La operación de particionar una lista consiste en que tomando como base una lista circular sencilla se construyen dos listas circulares sencillas pasando los elementos que se encuentren en una posición impar a la primera lista y los elementos que se encuentren en una posición par se pasan a la segunda lista. La solución de este problema se puede hacer de dos maneras:

Solución 1:

Crear cada uno de los nodos de las dos nuevas listas y copiar los datos que se encuentren en la lista original a la lista que le corresponda, y al final eliminar todos los nodos de la lista original.

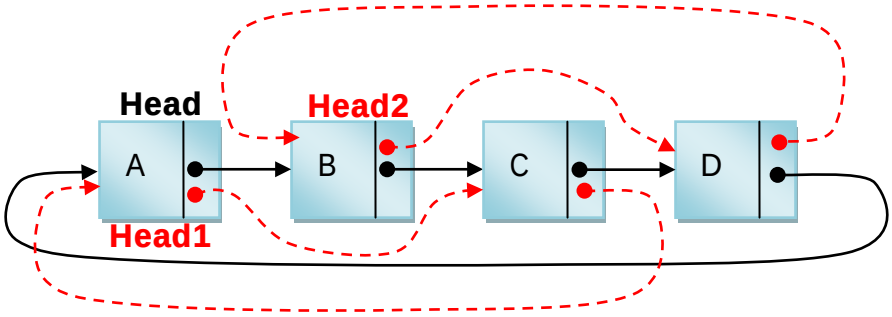






Solución 2:

En esta solución no es necesario crear y eliminar nodos, lo único que se hace es ligar todos los nodos que se encuentran en una posición impar para formar la primera de las listas y todos los nodos que se encuentran en una posición par para formar la segunda de las listas.



Solución en pseudocódigo

```
Si head <> null  y head(liga) <> head entonces
    head1 ← head
    head2 ← head(liga)
    p ← head1
    q ← head2
    i ← 0
    Mientras q(liga) <> head
        p(liga) ← q(liga)
        p ← q
        q ← q(liga)
        i ← i +1
    Si (i%2 = 0) entonces
        p(liga) ← head1
        q(liga) ← head2
    De lo contrario
        p(liga) ← head2
        q(liga) ← head1
De lo contrario
    Mensaje ('No hay suficientes nodos para particionar...')
```

Código en C++

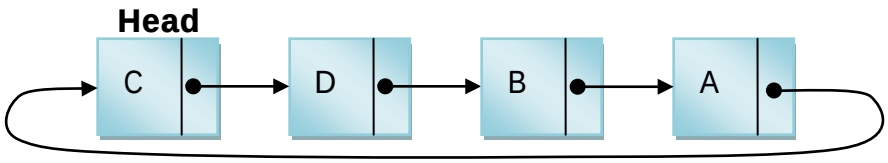
```
void lista_sencilla_circular::particionar(
lista_sencilla_circular &a,lista_sencilla_circular &b)
{
   nodo p,q;
   int i;
   i = 0;
   if (head != NULL && head->liga != head)
   {
       p = head;
       q = head->liga;
       a.head = p;
       b.head = q;
       while (q->liga != head)
       {
            p->liga = q->liga;
            p = q;
            q = q->liga;
            i++;
       }
       if (i%2 == 0)
       {
            p->liga = a.head;
            q->liga = b.head;
       }
       else
       {
            p->liga = b.head;
            q->liga = a.head;
       }
   }
   else
       cout << "No hay suficientes nodos...";
}
```

3.18 Buscar un elemento

Diseñar un algoritmo que permita buscar un elemento x en una lista sencilla circular.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x, hasta que se encuentre el nodo con el valor de x o que se acabe la lista.



Solución en pseudocódigo

```
Si head <> null
    existe ← falso
    Leer (dato)
    p ← head
    Repite
        Si p(dato) = dato
            existe ← verdadero
        P ← p(liga)
    Hasta (p=head)
    Si existe = verdadero entonces
        Mensaje (Si se encuentra el elemento x)
    De lo contrario
        Mensaje (No se encuentra el elemento x)
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

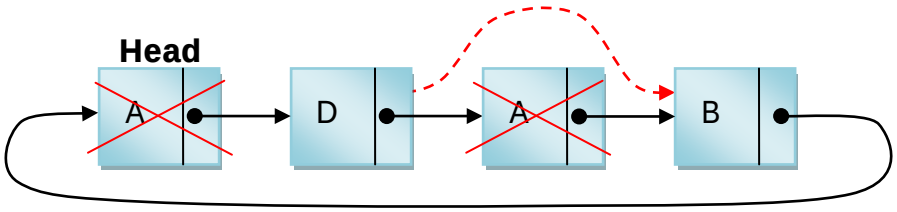
```
int lista_sencilla_circular::buscar(char valor)
{
   nodo p;
   int existe;
   if (head != NULL)
   {
       existe = false;
       p=head;
       do
       {
           if (p->dato == valor)
           {
               existe = true;
           }
           p = p->liga;
       } while (p!= head);
       if (existe)
           cout << "Si se encuentra el elemento";
       else
           cout << "No se encuentra el elemento";
   }
   else
       cout << "Lista vacía...";
   return (existe);
}
```

3.19 Eliminar repeticiones

Diseñar un algoritmo que permita eliminar todas las repeticiones del elemento x en una lista sencilla circular.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x. Si al comparar los elementos existe una coincidencia entonces se elimina el nodo, y es necesario ligar el nodo antecesor con el nodo sucesor. Se sugiere que en el ciclo del recorrido antes de que se avance al siguiente nodo, se deje otro apuntador en el nodo, para facilitar el ligado del nodo que se elimina.



Solución en pseudocódigo

```
Si head <> null
    p ← head
    Mientras p(liga) <> head
        Si p(dato) = dato entonces
            Si p = head entonces
                q ← head
                Mientras q(liga) <> head
                    q ← q(liga)
                Si q <> head entonces
                    head ← head (liga)
                    q(liga) ← head
                De lo contrario
                    head ← null
                Eliminar (p)
                p ← head
            De lo contrario
                r(liga) ← p(liga)
                Eliminar(p)
                p ← r(liga)
        De lo contrario
            r ← p
            p ← p(liga)
        Si p(dato) =  valor entonces
            q ← head
            Mientras q(liga) <> head
                q ← q(liga)
            q(liga) ← head
            Eliminar(p)
        De lo contrario
            head ← null
            Eliminar(p)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_sencilla_circular::eliminar_repetidos(char valor)
{
    nodo p,q,r;
    if (head != NULL)
    {
        p = head;
       while (p->liga != head)
        {
            if (p->dato == valor)
            {
                if (p==head)
                {
                    q = head;
                    while (q->liga != head)
                        q = q->liga;
                    if (q != head)
                    {
                        head = head->liga;
                        q->liga = head;
                    }
                    else
                        head = NULL;
                    delete(p);
                    p = head;
                }
                else
                {
                    r->liga = p->liga;
                    delete(p);
                    p = r->liga;
                }
            }
            else
            {
                r = p;
                p = p->liga;
            }
        }
        if (p->dato == valor)
        {
```

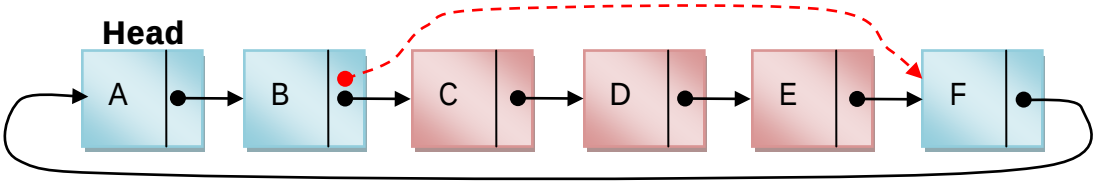
```
            if (p!= head)
            {
                q = head;
                while (q->liga != p)
                    q = q->liga;
                q->liga = head;
                delete(p);
            }
            else
            {
                head = NULL;
                delete(p);
            }
        }
    }
    else
        cout << "Lista vacía...";
}
```

3.20 Eliminar subcadena

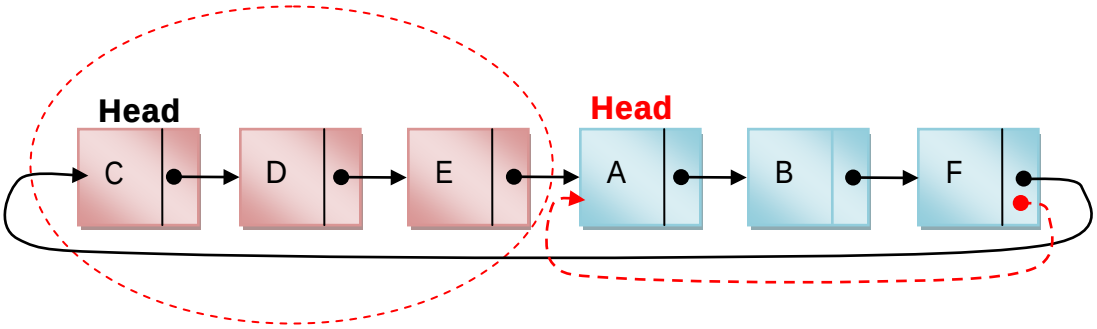
Diseñar un algoritmo que permita eliminar una subcadena de una lista sencilla circular.

Análisis del problema

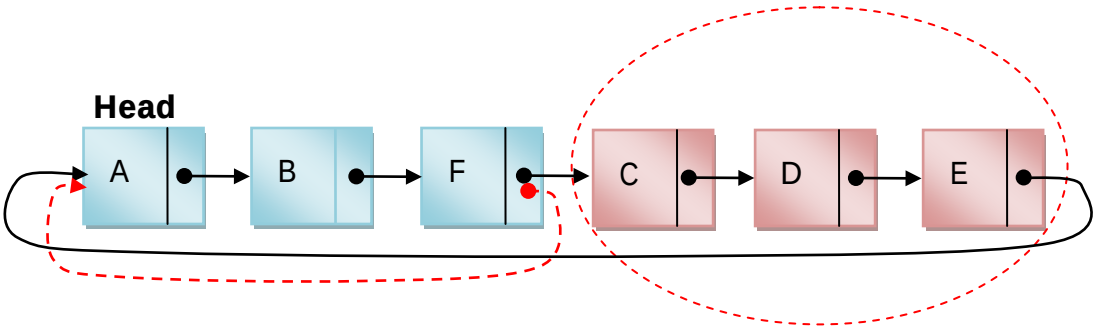
Para resolver este problema es necesario contar con el valor de la subcadena que se va a eliminar y calcular su tamaño. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación. Se localiza el inicio de la subcadena y se eliminan los nodos que forman parte de la subcadena. Es necesario validar si la subcadena está al inicio de la lista, porque en ese caso, es necesario mover la variable head al final de la subcadena.



Subcadena en una posición intermedia



Subcadena al inicio de la lista



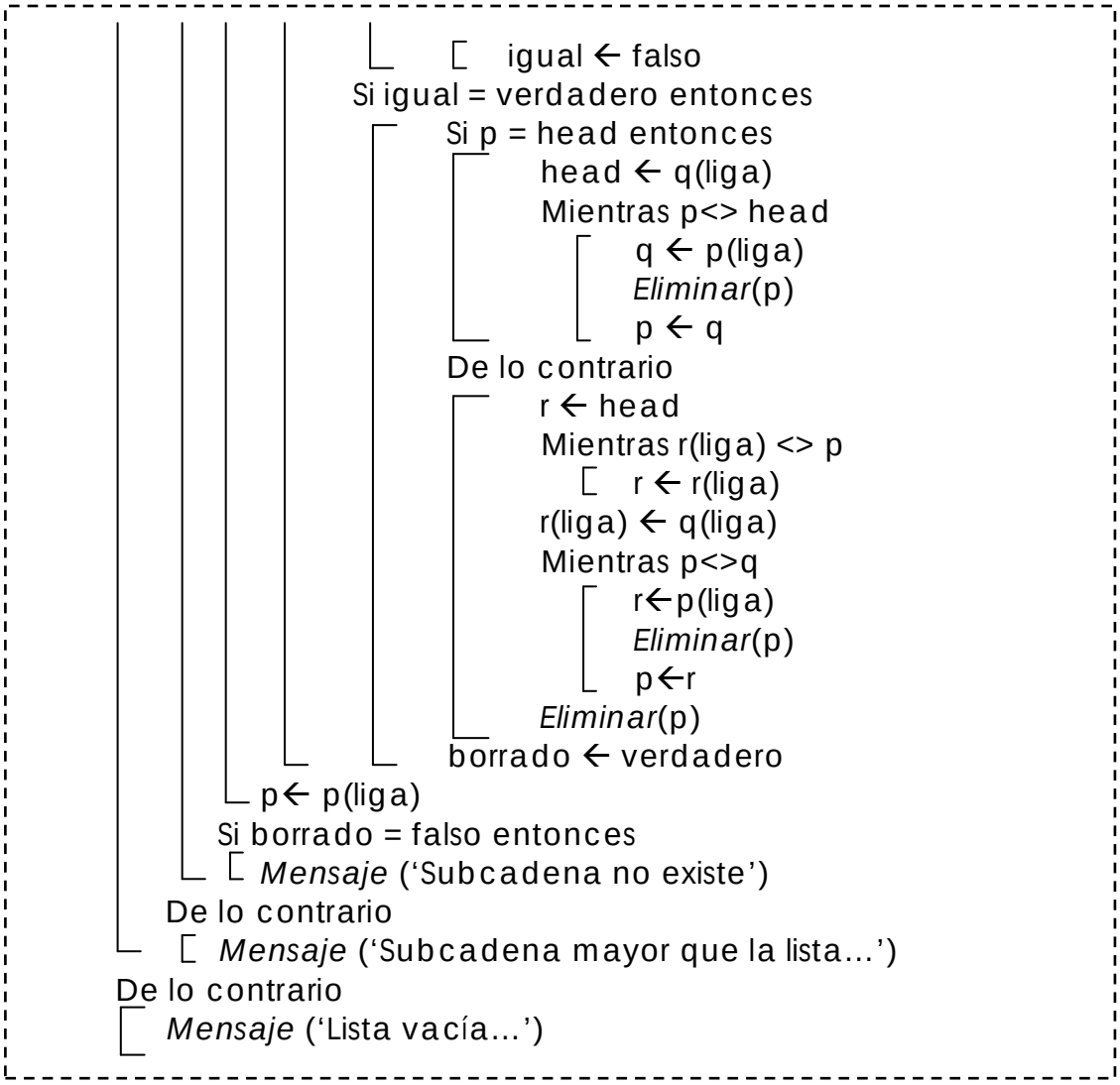
Subcadena al final de la lista

Solución en pseudocódigo

```
Si head <> null
    Leer (subcadena)
    c ← 1, p ← head, tam ← tamaño(subcadena)
    Mientras p(liga) <> head
        p ← p(liga)
        c ← c + 1
    Si c > tam entonces
        p← head, borrado ← falso
        Mientras (p(liga) <> head y (borrado = falso)
            Si  p(dato)  = subcadena[1] entonces
                q ← p, i←1, igual ← verdadero
                Mientras i < tam y igual = verdadero y q<> head
                    q ← q(liga)
                    i ← i+1
                    Si q <> head entonces
                        Si q(dato) <> subcadena[i] entonces
                            igual ← falso
                    De lo contrario
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay más de un elemento



Código en C++

```
void lista_sencilla_circular::eliminar_subcadena(char *valor)
{
    nodo p,q,r,s;
    int i,c,tam,borrado,igual;
    c = 1;
    p = head;
    tam = strlen(valor);
    if (head != NULL)
    {
        while (p->liga != head)
```

```
        {
            p = p->liga;
            c++;
        }
        s=p;
        if (c > tam)
        {
            p = head;
            borrado = false;
            while (p->liga!= head && borrado == false)
            {
                if (p->dato == valor[0])
                {
                    q = p;
                    i = 0;
                    igual = true;
                    while (i < tam-1 && igual && q!= s)
                    {
                        q = q->liga;
                        i++;
                        if (q != head)
                        {
                            if (q->dato != valor[i])
                                igual = false;
                        }
                        else
                            igual = false;
                    }
                    if (igual)
                    {
                        if (p== head)
                        {
                            head = q->liga;
                            while (p!=head)
                            {
                                q = p->liga;
                                delete(p);
                                p=q;
                            }
                            s->liga = head;
                        }
                        else
                        {
                            r = head;
```

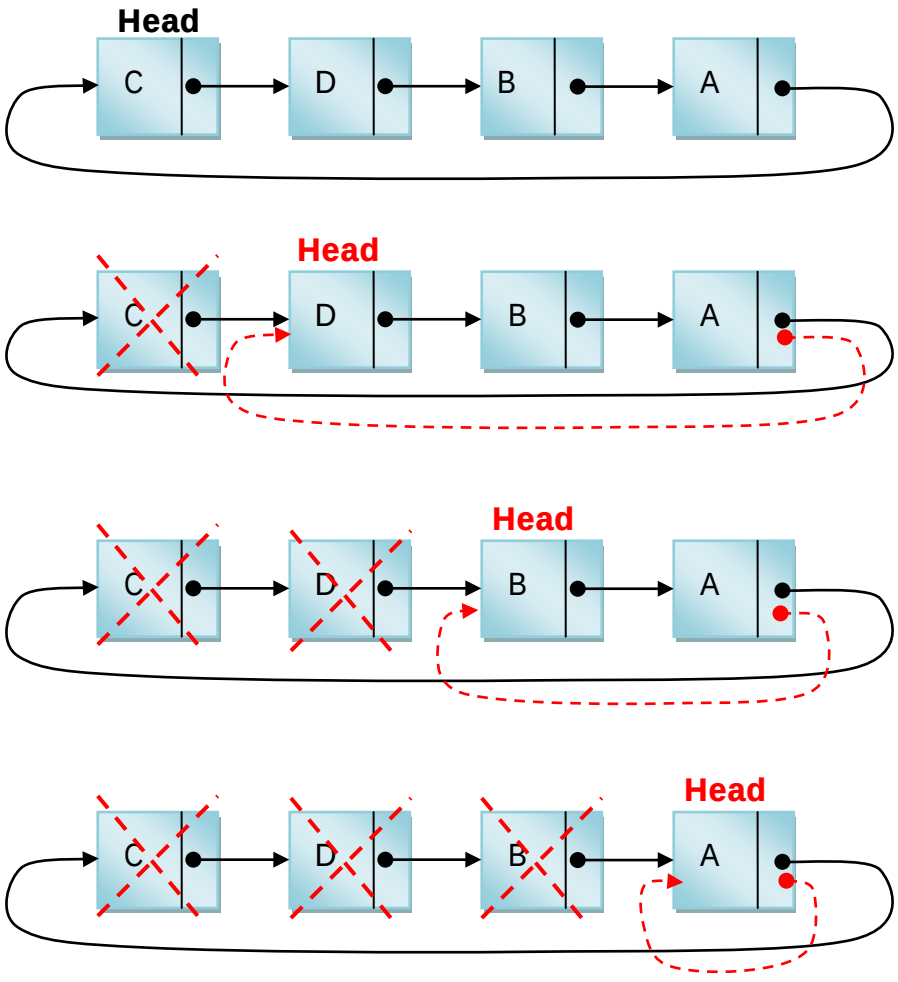
```
                                while (r->liga != p)
                                    r = r->liga;
                                r->liga = q->liga;
                                while (p!=q)
                                {
                                    r=p->liga;
                                    delete(p);
                                    p=r;
                                }
                                delete(p);
                            }
                            borrado=true;
                        }
                    }
                    p=p->liga;
                }
                if (borrado == false)
                    cout << "Subcadena no existe...";
            }
            else
                cout << "Subcadena mayor que la lista...";
        }
        else
            cout << "Lista vacía";
}
```

3.21 Borrar la lista

Diseñar un algoritmo que permita borrar todos los elementos de una lista sencilla circular.

Análisis del problema

Para resolver este problema es necesario recorrer todos los nodos de la lista desde el inicio para ir eliminando cada uno de los nodos de la lista y al final inicializar *head* en nulo.



Solución en pseudocódigo

```
Si head <> null
    q ← head
    Mientras q(liga) <> head
        q ← q(liga)
    Mientras head(liga) <> head
        head ← head(liga)
        q(liga) ← head
        Eliminar (p)
        p ← head
    head ← null
    Eliminar(p)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_sencilla_circular::eliminar()
{
    nodo p,q;
    if (head != NULL)
    {
        q = head;
        while (q->liga != head)
        {
            q = q->liga;
        }
        while (head->liga != head)
        {
            head = head->liga;
            q->liga = head;
            delete(p);
            p=head;
        }
        head = NULL;
        delete(p);
    }
    else
        cout << "Lista vacía...";
}
```

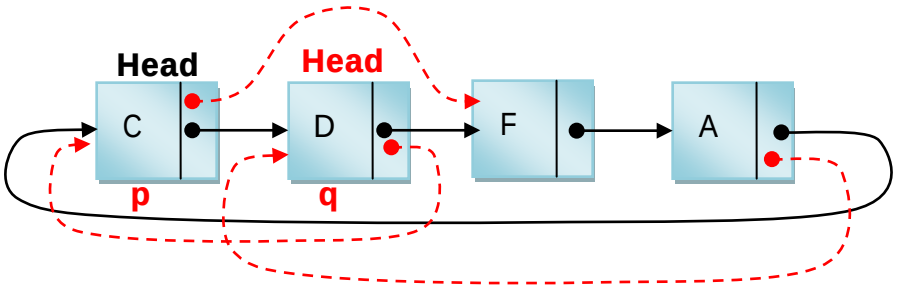
3.22 Ordenar burbuja intercambiando ligas

Diseñar un algoritmo que permita ordenar una lista sencilla circular con el método de la burbuja, intercambiando las ligas.

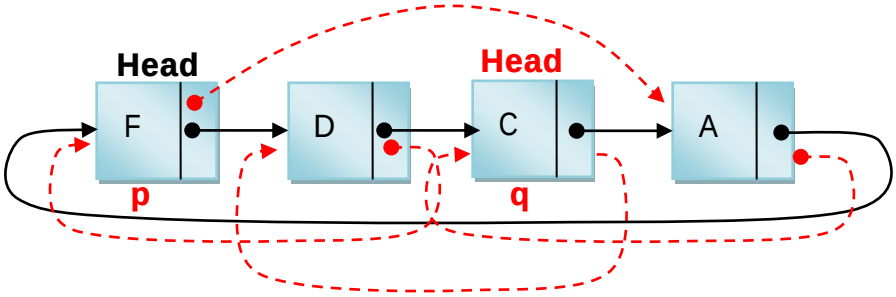
Análisis del problema

Para resolver este problema es necesario contar con al menos dos nodos. Al intercambiar las ligas los valores no cambiaran de localidad de memoria. El método de la burbuja funciona comparando el primer elemento de la lista con el resto de los elementos, en caso de que sea necesario se realiza el intercambio, posteriormente se avanza hacia el segundo elemento para compararlo con el resto y así sucesivamente hasta terminar de comparar todos los elementos. Es necesario considerar cuatro casos posibles cuando se realice un intercambio:

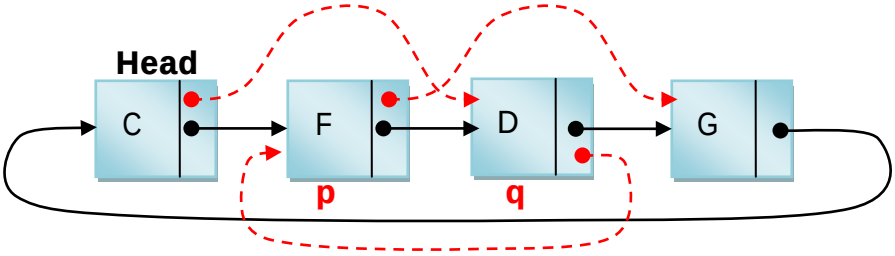
Caso 1: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.



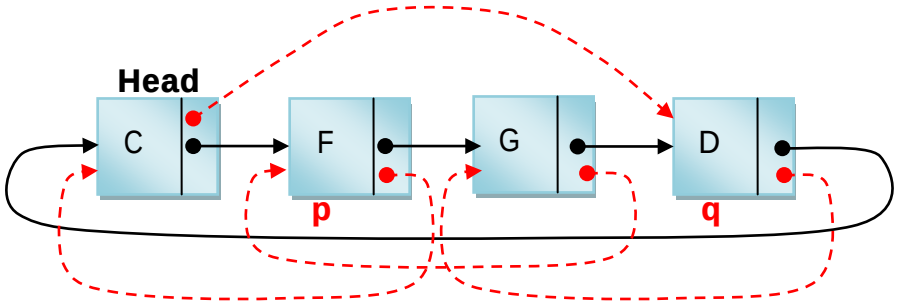
Caso 2: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Caso 3: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.

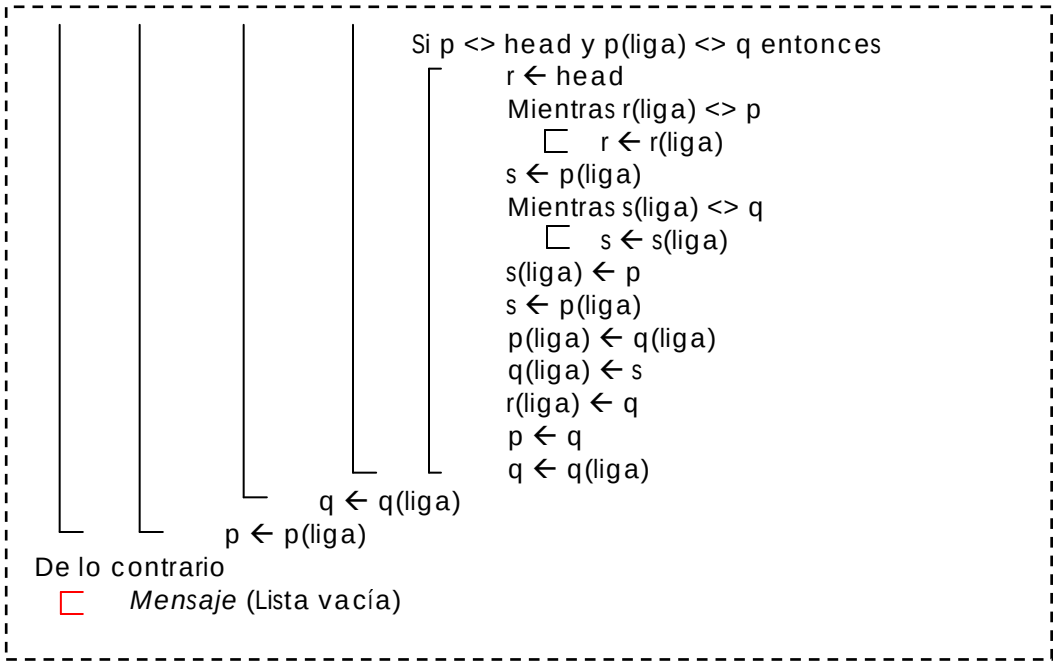


Caso 4: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras  p(liga) <>  head
        q ← p(liga)
        Mientras q <> head
            Si p(dato) > q(dato) entonces
                Si p = head y p(liga) = q entonces
                    r ← head
                    mientras (r(liga) <> head)
                        r ← r(liga)
                    p(liga) ← q(liga)
                    q(liga) ← p
                    head ← q
                    r(liga) ← head
                    p ← q
                    q ← q(liga)
                Si p = head y p(liga) <> q entonces
                    s ← head
                    Mientras s(liga) <> head
                        s ← s(liga)
                    r ← p(liga)
                    Mientras r(liga) <> q
                        r ← r(liga)
                    r(liga) ← p
                    r ← p(liga)
                    Si (s<>q) entonces
                        p(liga) ← q(liga)
                        q(liga) ← r
                        head ← q
                        s(liga) ← head
                        q ← p
                        p ← head
                    De lo contrario
                        p(liga) ← q
                        q(liga) ← r
                        head ← q
                        q ← p
                        p ← head
                Si p <> head y p(liga) = q entonces
                    r ← head
                    Mientras r(liga) <> p
                        r ← r(liga)
                    r(liga) ← q
                    p(liga) ← q(liga)
                    q(liga) ← p
                    p ← q
                    q ← q(liga)
```



Código en C++

```
void lista_sencilla_circular::burbuja_ligas()
{
    nodo p,q,r,s;
    if (head != NULL)
    {
      p = head;
      while (p->liga != head)
      {
          q = p->liga;
          while (q != head)
          {
              if (p->dato > q->dato)
              {
                  if (p==head && p->liga==q)
                  {
                      r=head;
                      while (r->liga != head)
                          r=r->liga;
                      p->liga = q->liga;
                      q->liga = p;
                      head = q;
                      r->liga = head;
```

```
                    p = q;
                    q = q->liga;
                }
                else
                {
                  if (p==head && p->liga != q)
                  {
                    s=head;
                    while (s->liga != head)
                          s = s->liga;
                    r = p->liga;
                    while (r->liga != q)
                          r = r->liga;
                    r->liga = p;
                    r = p->liga;
                    if (s!=q)
                    {
                      p->liga = q->liga;
                      q->liga = r;
                      head = q;
                      s->liga = head;
                      q = p;
                      p = head;
                    }
                  else
                    {
                       p->liga = q;
                       q->liga = r;
                       head = q;
                       q = p;
                       p = head;
                    }
                  }
                  else
                  {
                    if (p!=head && p->liga == q)
                     {
                       r = head;
                       while (r->liga != p)
                          r = r->liga;
                       r->liga = q;
                       p->liga = q->liga;
                       q->liga = p;
                       p = q;
```

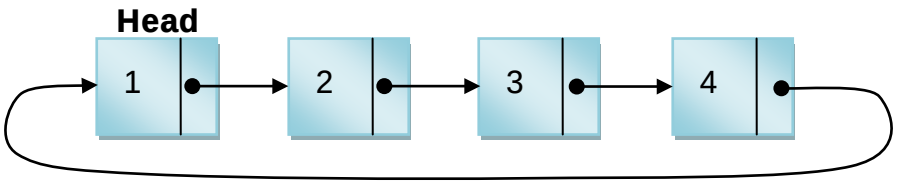
```
                        q = q->liga;
                    }
                else
                    {
                        if (p!=head && p->liga != q)
                        {
                            r = head;
                            while (r->liga != p)
                                r = r->liga;
                            s = p->liga;
                            while (s->liga != q)
                                s = s->liga;
                            s->liga = p;
                            s = p->liga;
                            p->liga = q->liga;
                            q->liga = s;
                            r->liga = q;
                            p = q;
                            q = q->liga;
                        }
                    }
                    }
                q = q->liga;
                }
            p = p->liga;
            }
    }
    else
        cout << "Lista vacía...";
}
```

3.23 Buscar posición

Diseñar un algoritmo que permita devolver en qué posición se encuentra un carácter en una lista sencilla circular.

Análisis del problema

Para resolver este problema es necesario contar con el valor del carácter que se buscará en la lista, recorrer toda la lista comparando el valor de cada nodo con el valor del carácter hasta encontrar el final de la lista o hasta que se encuentre el carácter.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (valor)
    p ← head
    pos ← 1
    Mientras (( p(liga) <>  head) o (p(dato)<>valor))
        p ← p(liga)
        pos ← pos + 1
    Si p(dato) = valor entonces
        Mensaje ('Posición = ', pos)
    De lo contrario
        Mensaje ('El valor no se encuentra en la lista')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

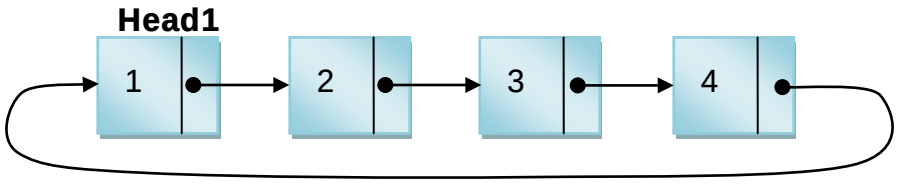
```
int lista_sencilla_circular::posicion(char valor)
{
   nodo p;
   int pos;
   if (head != NULL)
   {
       p = head;
       pos = 1;
       while (p->liga != head && p->dato != valor)
       {
           p = p->liga;
           pos++;
       }
       if (p->dato != valor)
          pos = 0;
    }
    return pos;
}
```

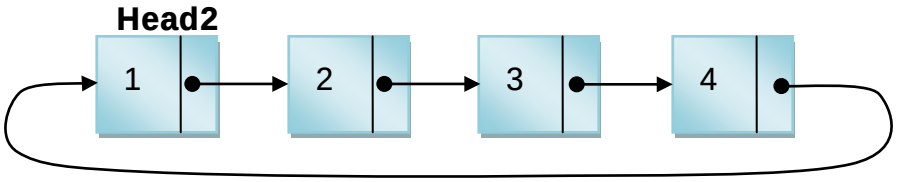
3.24 Comparar dos listas

Diseñar un algoritmo que permita comprobar si dos listas sencillas circulares son exactamente iguales.

Análisis del problema

Para resolver este problema es necesario recorrer ambas listas al mismo tiempo para ir comparando nodo por nodo su valor. Antes de iniciar el recorrido se puede verificar si la longitud de ambas listas es la misma, en caso de que no sea la misma longitud se puede asumir que las listas son diferentes.





Solución en pseudocódigo

```
Si head1 <> null y head2 <> null entonces
    Si tamaño(lista1) == tamaño(lista2) entonces
        p ← head1
        q ← head2
        igual ← verdadero
            Repite
                Si p(dato) <> q(dato) entonces
                    igual ← falso
                p ← p(liga)
                q ← q(liga)
            Hasta ((igual= falso) o (p=head) o ( q=head))
        De lo contrario
            Igual ← falso
        Si igual = verdadero entonces
            Mensaje ('Listas iguales')
        De lo contrario
            Mensaje ('Las listas no son iguales')
    De lo contrario
        Mensaje (Lista vacía)
```

Código en C++

```
int lista_sencilla_circular::comparar(
lista_sencilla_circular a)
{
   nodo p,q;
   int igual;
   if (head != NULL && a.head != NULL)
```

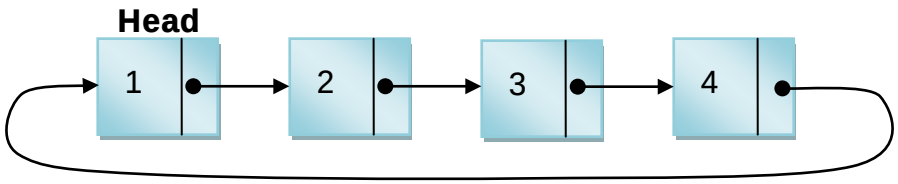
```
    {
      if (tamano() == a.tamano())
       {
        p = head;
        q = a.head;
        igual = true;
        do
        {
            if (p->dato != q->dato)
                igual = false;
            p = p->liga;
            q = q->liga;
        }while (igual && (p!= head && q!=head));
       }
      else
         igual = false;
    }
    else
        cout << "Listas vacías...";
    return igual;
}
```

3.25 Reemplazar texto

Diseñar un algoritmo que permita reemplazar parte de una lista sencilla circular por otro texto.

Análisis del problema

Para resolver este problema es necesario calcular el tamaño de la lista y el tamaño del texto que se va a reemplazar, para determinar si es posible que a partir de la posición que se indique se pueda llevar a cabo la operación. En caso de que los valores sean correctos, se ubica un apuntador en la posición en donde se hará el reemplazo y se inicia con el reemplazo.



Solución en pseudocódigo

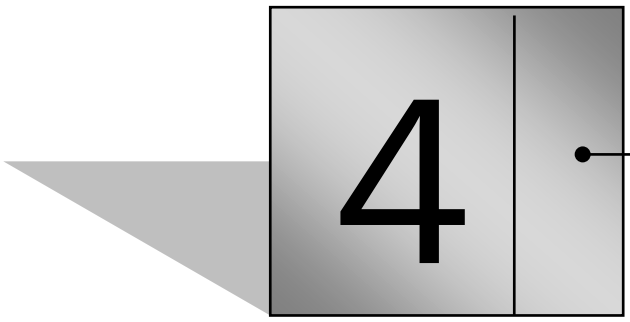
```
Si head <> null
    Leer (pos)
    Leer (texto)
    n ← tamaño(texto)
    c ← 1
    p ← head
    Mientras p(liga) <> head
        p ← p(liga)
        c ← c + 1
    Si (pos+n-1) <= c entonces
        p← head
        i = 0
        Mientras i <> pos entonces
            i ← i + 1
            p ← p(liga)
        j ← 1
        Mientras j <> n
            p(dato) ← texto[j]
            p ← p(liga)
            j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores...')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_sencilla_circular::reemplazar(int pos, char *valor)
{
    nodo p;
    int c,i,j,n;
    if (head != NULL)
    {
        n = strlen(valor);
```

```
        c=1;
        p = head;
        while (p->liga != head)
        {
            p = p->liga;
            c++;
        }
        if ((pos+n-1) <= c )
        {
            p = head;
            i = 1;
            while (i!= pos)
            {
                i++;
                p = p->liga;
            }
            j = 0;
            while (j!= n)
            {
                p->dato = valor[j];
                p = p->liga;
                j++;
            }
        }
        else
            cout << "Error en los valores";
    }
    else
        cout << "Lista vacía";
}
```

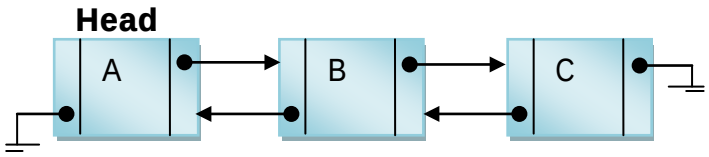
[Capítulo]



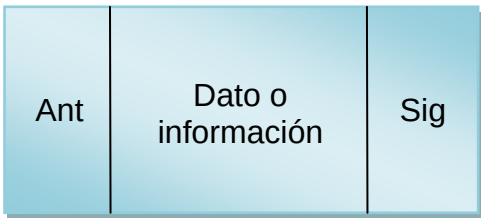
Listas Dobles Lineales

4 Listas Dobles Lineales

La característica principal de una lista doble lineal es que las ligas del último nodo y del primer nodo apuntan hacia el valor nulo.



El nodo de una lista doble lineal debe contener como mínimo tres campos: uno para almacenar la información y otros dos para guardar la dirección de memoria del nodo antecesor y sucesor. En la figura se puede apreciar la estructura del nodo para una lista doble.



Para definir la estructura del nodo en C++ se hace lo siguiente:

```
struct apuntador
{
  char dato;
  apuntador *sig;
  apuntador *ant;
};
```

Para simplificar la asignación de memoria se utiliza la siguiente función:

```
nodo nuevo()
{
 nodo p;
 p = new struct apuntador;
 return p;
}
```

Se presenta la clase lista_doble_lineal, la cual incluye la variable head y los métodos de las operaciones que se desarrollan en este capítulo para el manejo de las listas dobles lineales.

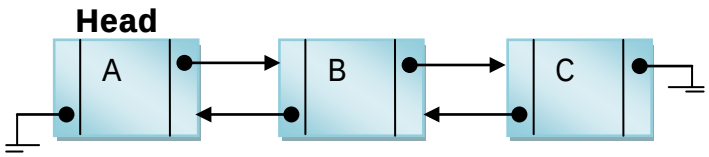
```
class lista_doble_lineal
{
  nodo head;
  public:
   lista_doble_lineal();
   void crear();
   void desplegar();
   void mayusculas();
   int tamano();
   void insertar_final();
   void insertar_inicio();
   void insertar(int posicion);
   void borrar_ultimo();
   void borrar_inicio();
   void borrar(int posicion);
   void desplegar_invertida();
   void burbuja();
   void invertir();
   void concatenar(lista_doble_lineal &b);
   void eliminar_subcadena(int n, int x);
   void intercalar(lista_doble_lineal &a);
   void particionar(lista_doble_lineal &a,
        lista_doble_lineal &b);
   int buscar(char valor);
   void eliminar_repetidos(char valor);
   void eliminar();
   int posicion(char valor);
   int comparar(lista_doble_lineal a);
   void burbuja_ligas();
   void reemplazar(int pos, char *valor);
   void eliminar_subcadena(char *valor);
};
```

4.1 Crear una lista

Diseñar un algoritmo que permita crear una lista doble lineal con n número de nodos.

Análisis del problema

Para resolver este problema es necesario la utilización de un ciclo que estará generando cada uno de los nodos que formaran parte de la lista. Es necesario introducir la información de cada uno de los nodos dentro del ciclo. Cada nodo se tiene que ligar con el nodo siguiente y con el nodo antecesor. La liga hacia el antecesor del primer nodo se le asigna el valor nulo y la liga hacia el sucesor del último nodo también se le asigna el valor nulo.



Solución en pseudocódigo

```
Si head = null entonces
    Repite
        Nuevo (p)
        Leer (p(dato))
        p(sig) ← null
        Si head = null entonces
            head ← p
            p(ant) ← null
        De lo contrario
            p(ant) ← q
            q(sig) ← p
        q ← p
        Leer (otro)
    Hasta (otro = NO)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

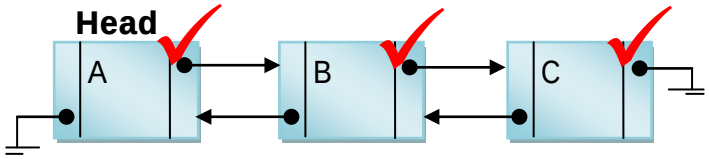
```
void lista_doble_lineal::crear()
{
  nodo p,q;
  char otro;
  do
  {
    p=nuevo();
    cout << "p(dato) = ";
    cin >> p->dato;
    p->sig = NULL;
    if (head == NULL)
    {
        head = p;
        p->ant = NULL;
    }
    else
    {
        p->ant = q;
        q->sig = p;
    }
    q=p;
    cout << "Capturar otro nodo s/n ? " ;
    cin >> otro;

  } while (otro == 's');
}
```

4.2 Recorrer una lista

Diseñar un algoritmo que permita desplegar el contenido de todos los nodos de una lista doble lineal.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se recorre toda la lista desde el primer nodo donde se encuentra *head* hasta encontrar el valor de nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Repite
        Desplegar (p(dato))
        p ← p(sig)
    Hasta (p= null)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_lineal::desplegar()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    while (p!=NULL)
    {
       cout << p->dato;
       p = p->sig;
    }
  }
```

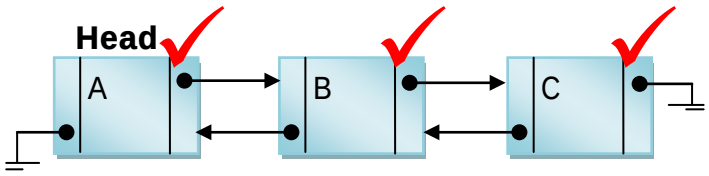
```
   else
      cout <<  "Lista Vacía";
}
```

4.3 Calcular tamaño

Diseñar un algoritmo que permita determinar el tamaño de una lista doble lineal.

Análisis del problema

Para calcular el tamaño de la lista es necesario recorrer todos los nodos de la lista desde el primer nodo hasta encontrar el valor de nulo. Para contar el total de nodos se utiliza un contador que se va incrementando.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    total ← 1
    Mientras p(sig) <> null
        p ← p(sig)
        total ← total + 1
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

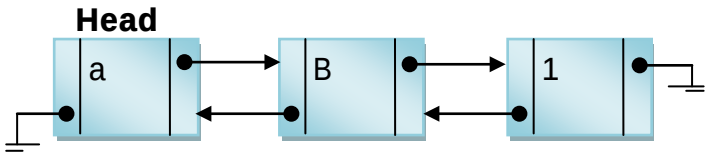
```
int lista_doble_lineal::tamano()
{
  nodo p;
  int total;
  if (head != NULL)
  {
    p = head;
    total = 1;
    while (p->sig !=NULL)
    {
       p = p->sig;
       total++;
    }
  }
  else
    cout <<  "Lista Vacía";
  return total;
}
```

4.4 Convertir a mayúsculas

Diseñar un algoritmo que permita convertir todos los elementos alfabéticos de una lista doble lineal de minúsculas a mayúsculas.

Análisis del problema

Para resolver este problema es necesario recorrer toda la lista desde el inicio, para ir comparando el valor del nodo y en caso de que sea una letra convertirla a mayúscula.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p <> null
        Si p(dato) es una letra entonces
            p(dato) ← mayúscula(p(dato))
        p ← p(sig)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

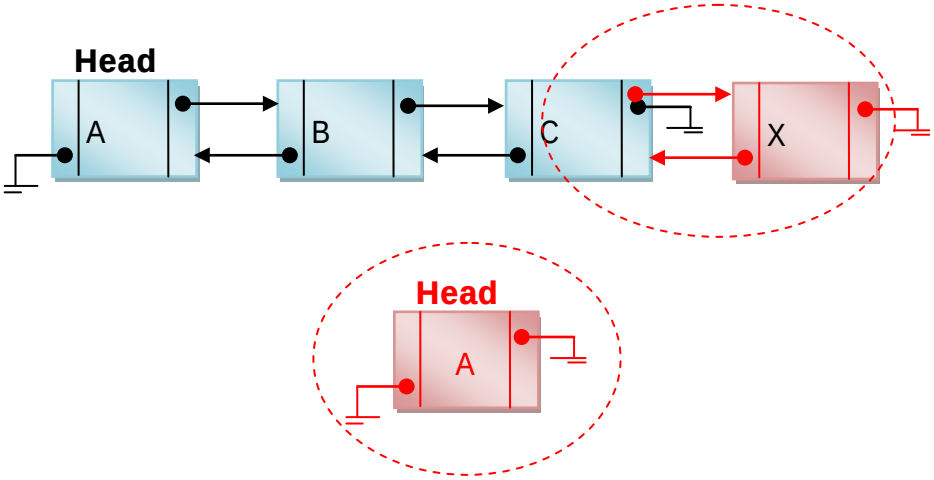
```
void lista_doble_lineal::mayusculas()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    while (p!=NULL)
    {
       if (p->dato>='a' && p->dato <= 'z')
            p->dato -=32;
       p = p->sig;
    }
  }
  else
    cout <<  "Lista Vacía";
}
```

4.5 Insertar al final

Diseñar un algoritmo que permita Insertar un nodo al final de una lista doble lineal.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía es necesario recorrer toda la lista para ubicarse en el último nodo, crear el nuevo nodo y ligar el nuevo nodo con el último nodo y el último nodo con el nuevo nodo. Si la lista está vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo y se les asigna el valor nulo a las ligas.



Solución en pseudocódigo

```
Nuevo(p)
Leer (p(dato))
p(sig) ← null
Si head <> null entonces
    q ← head
    Mientras q(sig) <> null
        q ← q(sig)
    q(sig) ← p
    p(ant) ← q
De lo contrario
    head ← p
    p(ant) ← null
```

Código en C++

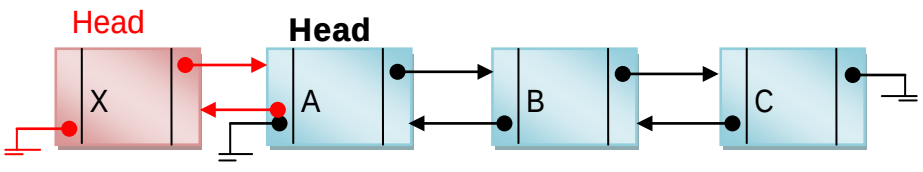
```
void lista_doble_lineal::insertar_final()
{
    nodo p,q;
    p = nuevo();
    cout << "p(dato) = ";
    cin >> p->dato;
    p->sig= NULL;
    if (head != NULL)
    {
          q = head;
          while (q->sig != NULL)
              q = q->sig;
          q->sig = p;
          p->ant = q;
    }
    else
    {
          head = p;
          p->ant = NULL;
    }
}
```

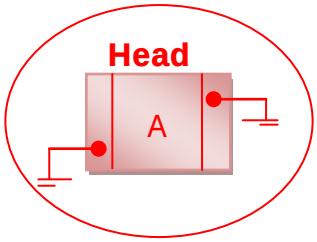
4.6 Insertar al inicio

Diseñar un algoritmo que permita Insertar un nodo al inicio de una lista doble lineal.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se crea el nuevo nodo y se liga con el primer nodo de la lista y *head* se mueve al nuevo nodo. Si la lista está vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo y se les asigna el valor de nulo a las ligas.





Solución en pseudocódigo

```
Nuevo(p)
Leer (p(dato))
p(ant) ← null
p(sig) ← head
Si head <> null entonces
   [    head(ant) ← p
head ← p
```

Código en C++

```
void lista_doble_lineal::insertar_inicio()
{
   nodo p;
   p = nuevo();
   cout << "p(dato) = ";
   cin >> p->dato;
   p->ant = NULL;
   p->sig = head;
   if (head !=  NULL)
     head->ant = p;
   head = p;
}
```

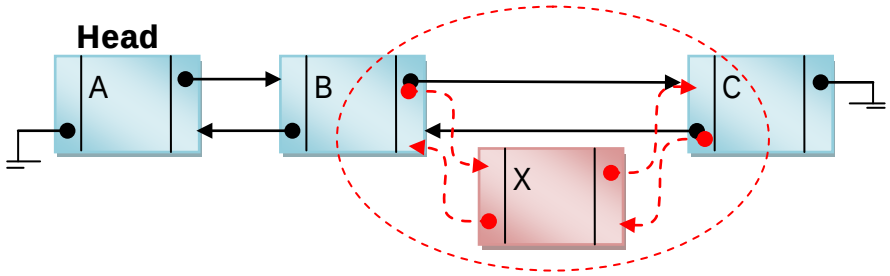
4.7 Insertar en cualquier posición

Diseñar un algoritmo que permita insertar un nodo en cualquier posición en una lista doble lineal.

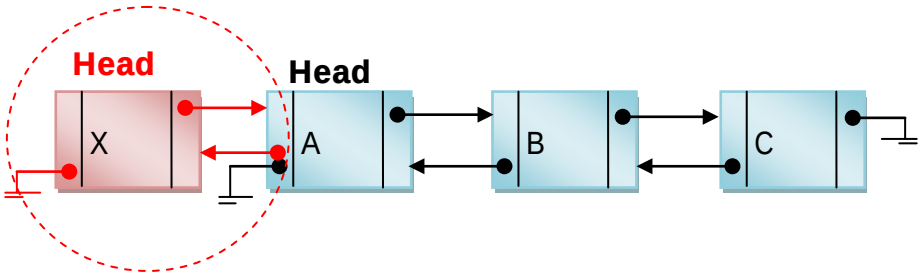
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos y si la posición en la que se desea insertar el nuevo es válida, es decir si es menor o igual al total de los nodos de la lista. Se deben consideran los casos siguientes:

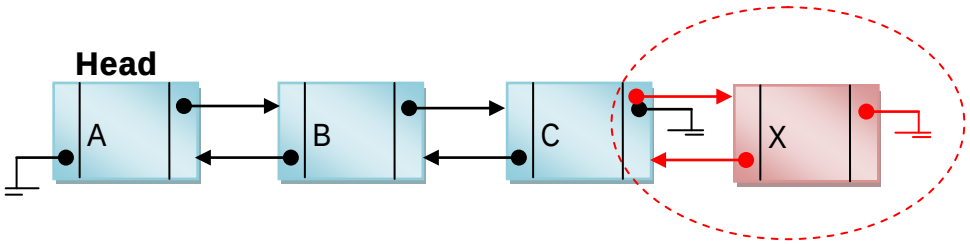
Caso 1: Insertar el nuevo nodo en una posición intermedia dentro de la lista.



Caso 2: Insertar el nuevo nodo al inicio de la lista. En este caso también se debe determinar si el nodo que se inserta es el primero de la lista, y en su caso, ubicar a head en dicho nodo.



Caso 3: Insertar el nuevo nodo al final de la lista.



Solución en pseudocódigo

```
Leer (posición)
p ← head
c ← 0
Mientras p <> null
    p ← p(sig)
    c ← c + 1
Si (posición  > 0) y (posición <= c+1) entonces
    Nuevo (p)
    Leer (p(dato))
    Si pos = 1 entonces
        p(sig) ← head
        p(ant) ← null
        p(sig(ant)) ← p
        head ← p
    De lo contrario
        q ← head
        para i = 1 hasta pos – 1
            q ← q(sig)
        p(sig) ← q(sig)
        p(ant) ← q
        q(sig) ← p
        Si p(sig) <> null entonces
            p(sig(ant)) ← p
De lo contrario
    Mensaje ('Posición incorrecta...')
```

Código en C++

```
void lista_doble_lineal::insertar(int posicion)
{
   nodo p,q;
   int c,i;
   p = head;
   c = 0;
   while (p != NULL)
   {
       p = p->sig;
       c++;
```

```
    }
    if ((posicion > 0) && (posicion <= c+1))
    {
        p = nuevo();
        cout << "p(dato) = ";
        cin >> p->dato;
        if (posicion==1)
        {
            p->sig = head;
            p->ant = NULL;
            p->sig->ant = p;
            head = p;
        }
        else
        {
            q = head;
            for(i=2; i<=posicion-1; i++)
                q = q->sig;
            p->sig = q->sig;
            p->ant = q;
            q->sig = p;
            if (p->sig != NULL)
                p->sig->ant = p;
        }
    }
    else
         cout <<  "Posición Incorrecta...";
}
```

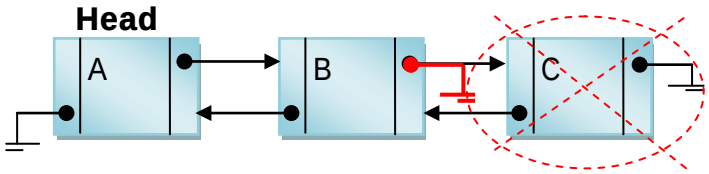
4.8 Borrar el último nodo

Diseñar un algoritmo que permita borrar el último nodo de una lista doble lineal.

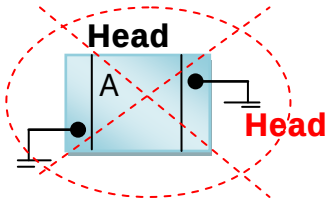
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene al menos un elemento. Es necesario recorrer los nodos de la lista para ubicarse en la penúltima posición y eliminar el último nodo. En el caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *Head* en nulo.

Caso 1: La lista contiene al menos dos elementos y se elimina el último.



Caso 2: La lista contiene un solo nodo. En este caso es necesario inicializar el valor de head a nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p← head
    Mientras p(sig) <> null
        p ← p(sig)
    Si  p = head entonces
        head ← null
    De lo contrario
        p(ant(sig)) ← null
    Eliminar(p)
De lo contrario
    Mensaje ('Lista vacía')
```

Código en C++

```
void lista_doble_lineal::borrar_ultimo()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    while (p->sig != NULL)
    {
        p = p->sig;
    }
    if (p == head)
        head = NULL;
    else
        p->ant->sig = NULL;
    delete(p);
  }
  else
    cout <<  "Lista Vacía";
}
```

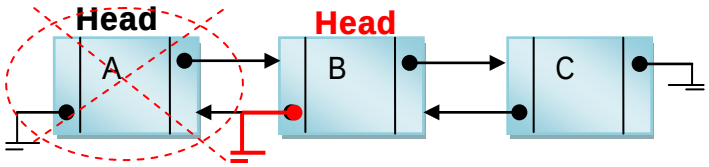
4.9 Borrar el primer nodo

Diseñar un algoritmo que permita borrar el primer nodo de una lista doble lineal.

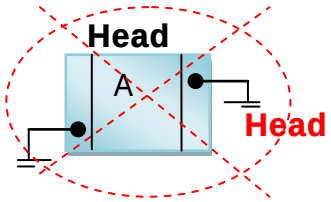
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene al menos un elemento. Si la lista contiene elementos se elimina el primer nodo de la lista, la variable *Head* se mueve al siguiente nodo de la lista y se actualiza el valor de anterior en nulo. En el caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *Head* en nulo.

Caso 1: La lista contiene al menos dos elementos. En este caso se elimina el primero y el head debe avanzar a la siguiente posición.



Caso 2: La lista contiene un solo nodo. En este caso es necesario inicializar el valor de head a nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p← head
    head ← head(sig)
    Si head <> null entonces
        head(ant)← null
    Eliminar(p)
De lo contrario
    Mensaje ('Lista vacía')
```

Código en C++

```
void lista_doble_lineal::borrar_inicio()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    head = head->sig;
    if (head != NULL)
        head->ant = NULL;
    delete(p);
  }
  else
    cout <<  "No hay elementos";
}
```

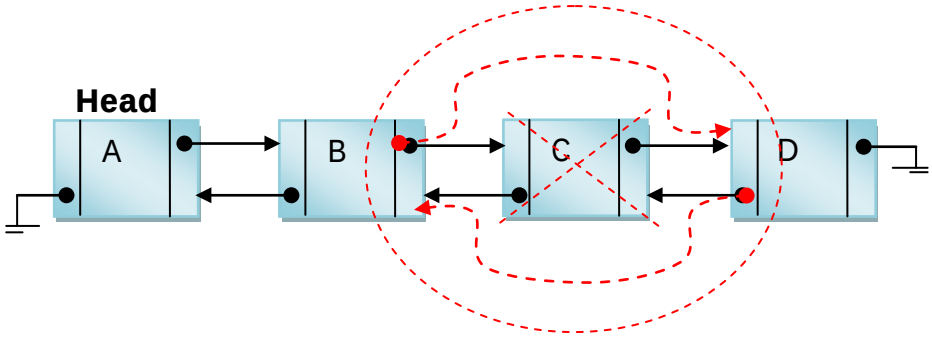
4.10 Borrar en nodo en cualquier posición

Diseñar un algoritmo que permita borrar un nodo en cualquier posición en una lista doble lineal.

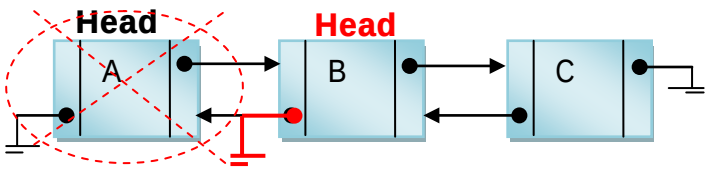
Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos y la posición del elemento que se desea eliminar es válida. En el caso de que la lista contenga únicamente un nodo, se elimina el nodo y se inicializa la variable *Head* en nulo. Si la posición es la última o la primera se utilizan los algoritmos para eliminar el último o el primer nodo. Si la posición es intermedia es necesario ligar el nodo antecesor y el sucesor del nodo que se elimina.

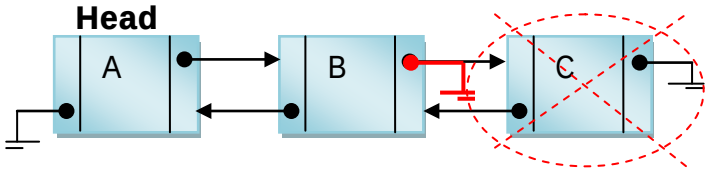
Caso 1: Borrar un nodo que se encuentre en una posición intermedia.



Caso 2: Borrar el primer nodo de la lista.



Caso 3: Borrar el último nodo de la lista.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (pos)
    p ← head
    c ← 1
    Mientras (c <> pos) y (p <> null)
        p ← p(sig)
        c ← c + 1
    Si c = pos entonces
        Si p = head entonces
            head ← head (sig)
            Si head <> null entonces
                head(ant) ← null
        De lo contrario
            p(ant(sig)) ← p(sig)
            Si p(sig) <> null entonces
                p(sig(ant)) ← p(ant)
        Eliminar (p)
    De lo contrario
        Mensaje ('Posición inválida...')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_lineal::borrar(int posicion)
{
 nodo p;
 int c;
 if (head != NULL)
 {
   p = head;
   c = 1;
   while ((c != posicion) && (p != NULL))
   {
       p = p->sig;
       c++;
   }
   if (c==posicion)
   {
```

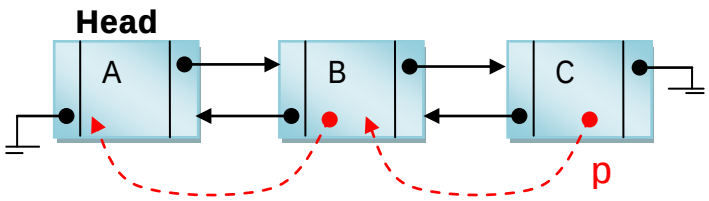
```
        if (p==head)
        {
             head = head->sig;
             if (head != NULL)
                 head->ant = NULL;
        }
        else
        {
             p->ant->sig = p->sig;
             if (p->sig != NULL)
                 p->sig->ant = p->ant;

        }
        delete (p);
    }
    else
        cout <<  "Posición inválida...";
  }
  else
    cout <<  "Lista Vacía";
}
```

4.11 Desplegar invertida

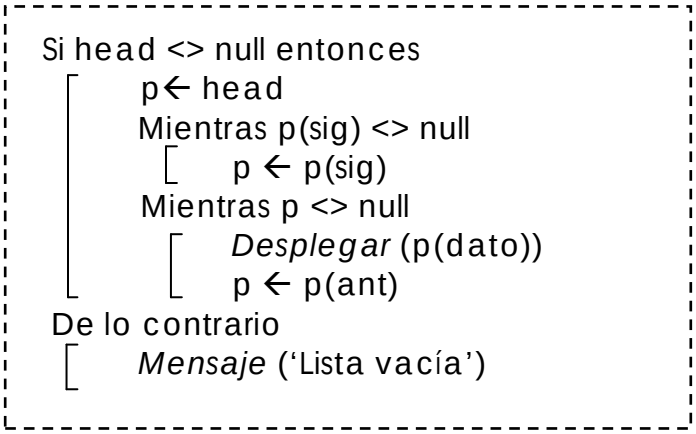
Diseñar un algoritmo que permita desplegar el contenido de una lista doble lineal de forma inversa.

Análisis del problema

Para resolver este problema utilizando listas dobles lineales es necesario ubicar un apuntador en el último nodo de la lista y a partir de esa posición recorrer la lista hasta llegar al primer nodo. Se tiene que verificar si la lista contiene nodos. Esta operación se vuelve más sencilla en una lista doble que en una sencilla debido a que cada nodo apunta además del nodo sucesor al nodo antecesor.



Solución en pseudocódigo



Código en C++

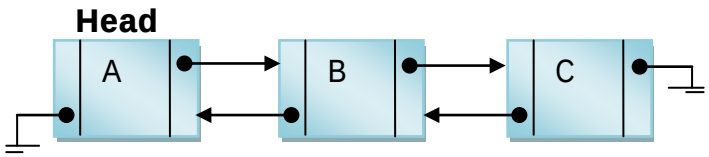
```
void lista_doble_lineal::desplegar_invertida()
{
   nodo p;
   if (head != NULL)
   {
      p = head;
      while (p->sig != NULL)
        p = p->sig;
      while (p != NULL)
      {
          cout << p->dato;
          p = p->ant;
      }
   }
   else
      cout <<  "Lista Vacía";
}
```

4.12 Ordenar burbuja

Diseñar un algoritmo que permita ordenar una lista doble lineal utilizando el método de la burbuja.

Análisis del problema

Para la implementación del método de ordenación de la burbuja se requieren dos ciclos anidados para ir comparando los elementos y hacer los intercambios que sean necesarios.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(sig) <> null
        q ← p(sig)
        Mientras (q <> null)
            Si (q(dato < p(dato) entonces
                aux ← p(dato)
                p(dato) ← q(dato)
                q(dato) ← aux
            q ← q(sig)
        p ← p(sig)
De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

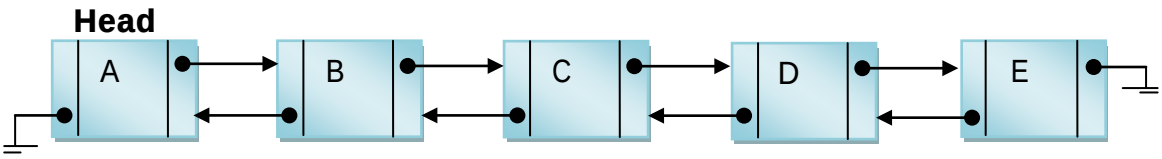
```
void lista_doble_lineal::burbuja()
{
  nodo p,q;
  char aux;
  if (head != NULL)
  {
      p = head;
      while (p->sig != NULL)
      {
          q = p->sig;
          while (q != NULL)
          {
              if (q->dato < p->dato)
              {
                  aux = p->dato;
                  p->dato = q->dato;
                  q->dato = aux;
              }
              q = q->sig;
          }
          p = p->sig;
      }
  }
  else
      cout <<  "No hay elementos";
}
```

4.13 Invertir una lista

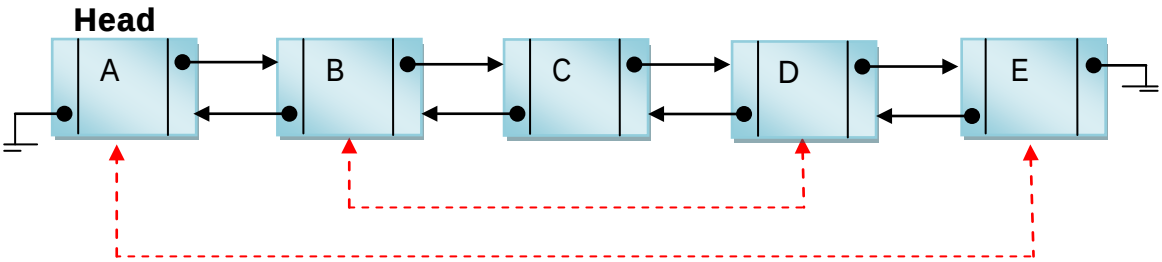
Diseñar un algoritmo que permita invertir los nodos de una lista doble lineal.

Análisis del problema

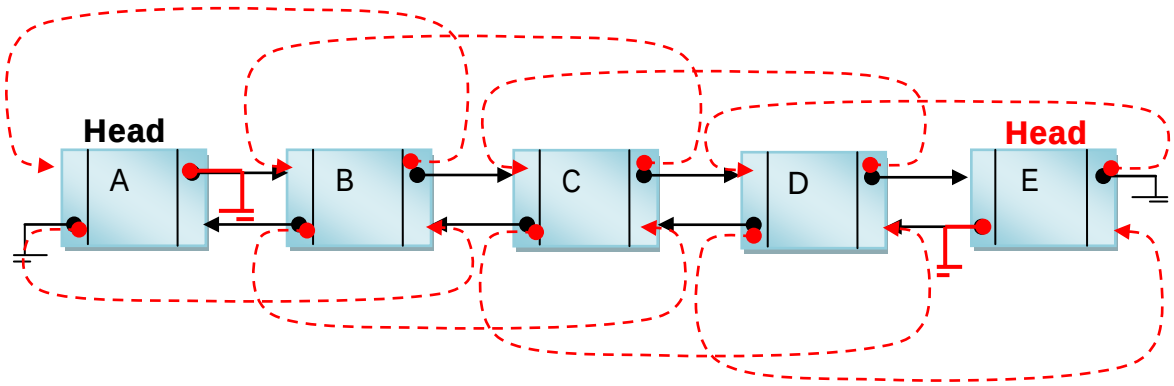
La operación de invertir una lista puede hacerse de dos maneras: la primera intercambiando únicamente los datos y la segunda dejando los datos en la posición de memoria en la que se encuentran y cambiar el ligado de los nodos para que la lista quede invertida.



Solución 1: Moviendo datos.



Solución 2: Moviendo ligas.



Solución en pseudocódigo

```
Si head(liga) <> null entonces
    p ← head
    Mientras p(sig) <> null
        p ← p(sig)
    head2 ← p
    Mientras (p <> null)
        q ← p(ant)
        aux ← p(sig)
        p(sig) ← p(ant)
        p(ant) ← aux
        p ← q
    head ← head2
De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

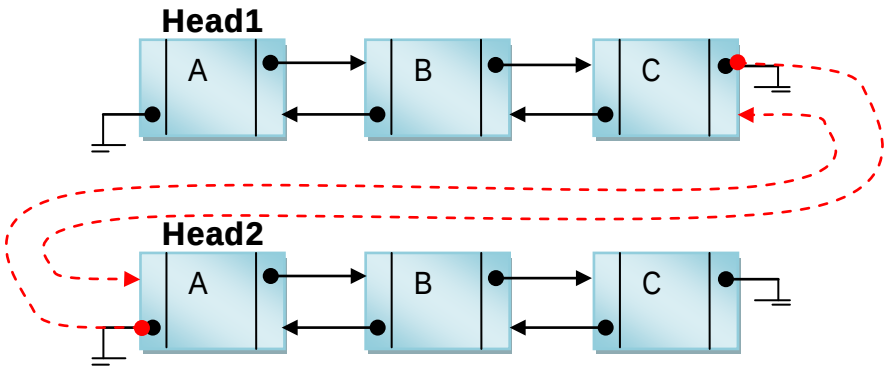
```
void lista_doble_lineal::invertir()
{
   nodo p,q,r,aux;
   if ((head != NULL) && (head->sig != NULL))
   {
       p = head;
       while (p->sig != NULL)
           p = p->sig;
       r = p;
       while (p != NULL)
       {
           q = p->ant;
           aux = p->sig;
           p->sig = p->ant;
           p->ant = aux;
           p = q;
       }
       head = r;
   }
   else
       cout <<  "No hay suficientes elementos";
}
```

4.14 Concatenar dos listas

Diseñar un algoritmo que permita concatenar los nodos de dos listas dobles lineales.

Análisis del problema

Para resolver este problema es necesario verificar que las dos listas contengan elementos. Se recorre la primera lista hasta el último nodo y se liga con el primer nodo de la segunda lista y la liga anterior del primer nodo de la segunda lista se liga con el último nodo de la primera lista.



Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
    p ← head1
    Mientras p(sig) <> null
        p ← p(sig)
    p(sig) ← head2
    head2(ant) ← p
    head ← head1
De lo contrario
    Mensaje ('No hay listas...')
```

Código en C++

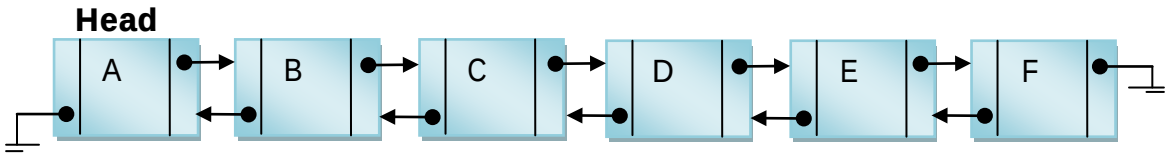
```
void lista_doble_lineal::concatenar(lista_doble_lineal &b)
{
   nodo p;
   if (head != NULL && b.head != NULL)
   {
        p = head;
        while (p->sig != NULL)
             p = p->sig;
        p->sig = b.head;
        b.head->ant = p;
        b.head = NULL;
   }
   else
        cout <<  "No hay listas";
}
```

4.15 Eliminar n número de nodos

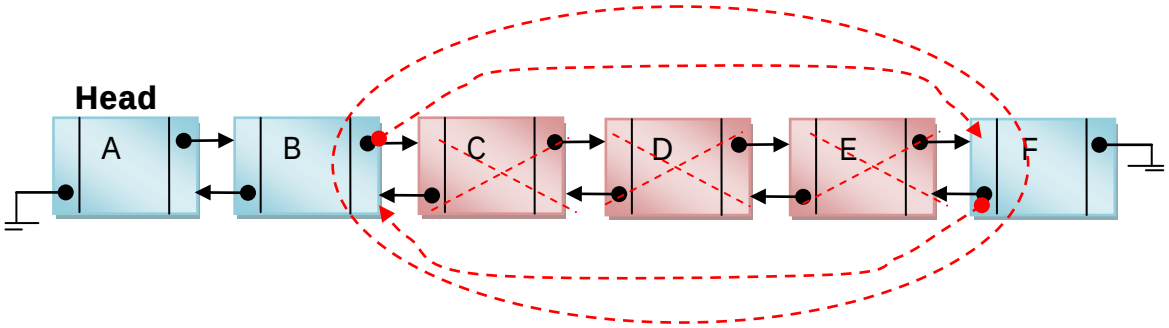
Diseñar un algoritmo que permita eliminar n número de nodos a partir de una posición x en una lista doble lineal.

Análisis del problema

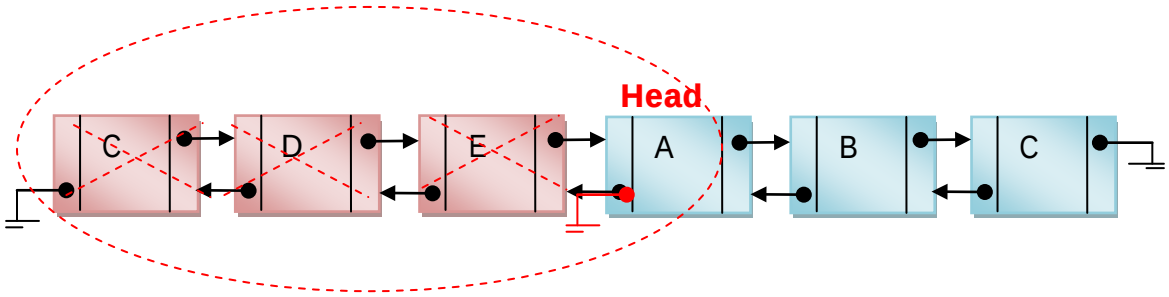
Para resolver este problema es necesario contar con el valor de n y x. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación, es decir, si existe la posición a partir de la cual se van a eliminar elementos y suficientes nodos para cumplir con el valor de n.



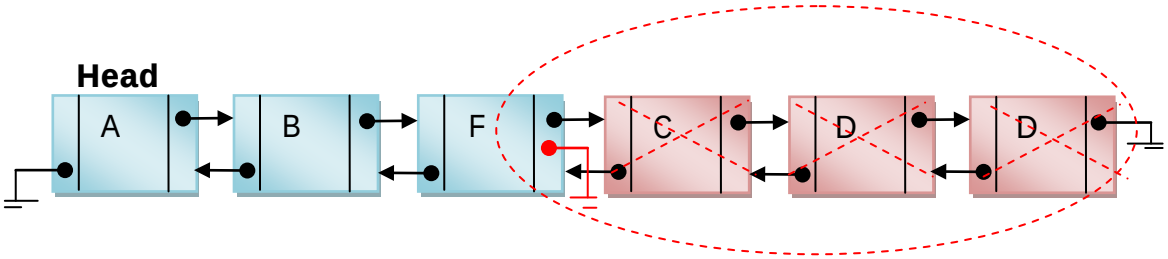
Caso 1: Los nodos a eliminar se encuentran en una posición intermedia dentro de la lista.



Caso 2: Los nodos a eliminar se encuentran al inicio de la lista.



Caso 3: Los nodos a eliminar se encuentran al final de la lista.



Solución en pseudocódigo

```
Si head <> null
    Leer (x,n)
    c ← 1
    p ← head
    Mientras p(sig) <> null
            p ← p(sig)
            c ← c + 1
    Si (x+n) <= c entonces
        p← head
        Si  x = 1 entonces
            j ← 0
            Mientras j <> n
                head ← head(sig)
                head(ant) ← null
                Eliminar(p)
                p ← head
                j ← j + 1
        De lo contrario
            i← 2
            Mientras i <> x entonces
                i ← i + 1
                p ← p(sig)
            j ← 1
            Mientras j <> n
                q ← p(sig)
                p(sig) ← q(sig)
                Si q(sig) <> null entonces
                        q(sig(ant)) ← p
                Eliminar(q)
                j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_lineal::eliminar_subcadena(int n, int x)
{
    nodo p,q;
    int c,i,j;
    if (head != NULL)
    {
        c = 1;
        p = head;
        while (p->sig != NULL)
        {
            p = p->sig;
            c++;
        }
        if ((x+n-1)<=c)
        {
            p = head;
            if (x==1)
            {
                j = 0;
                while (j!= n)
                {
                    head = head->sig;
                    head->ant = NULL;
                    delete(p);
                    p=head;
                    j++;
                }
            }
            else
            {
                i = 2;
                while (i!=x)
                {
                    i++;
                    p = p->sig;
                }
                 j = 0;
                while (j!=n)
                {
                    q = p->sig;
                    p->sig = q->sig;
                    if (q->sig != NULL)
```

```
                        q->sig->ant = p;
                    delete(q);
                    j++;
                }
            }
        }
        else
            cout << "Error en los valores";
    }
    else
        cout << "Lista vacía";
}
```

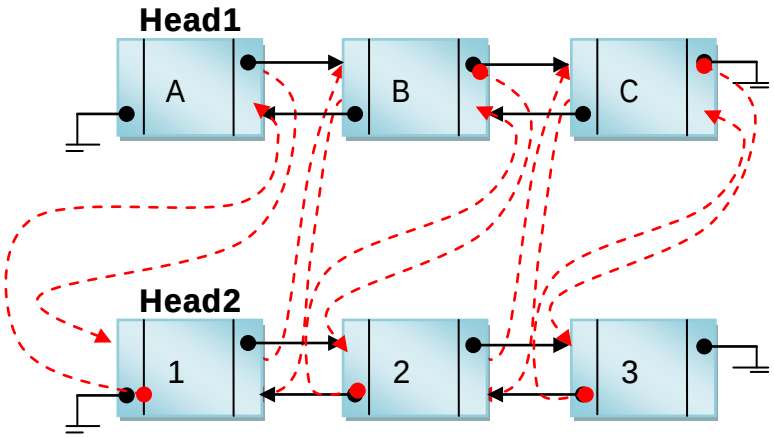
4.16 Intercalar dos listas

Diseñar un algoritmo que permita Intercalar los nodos de dos listas dobles lineales.

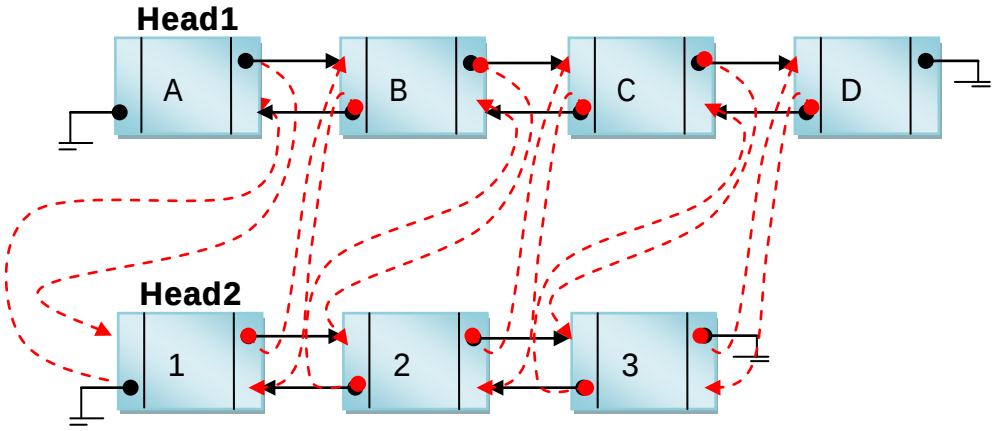
Análisis del problema

La operación de intercalar dos listas se lleva a cabo recorriendo ambas listas al mismo tiempo. Es necesario verificar si las dos listas contienen elementos y comparar el tamaño de cada una de las listas. La solución debe contemplar que las listas no necesariamente tendrán el mismo tamaño.

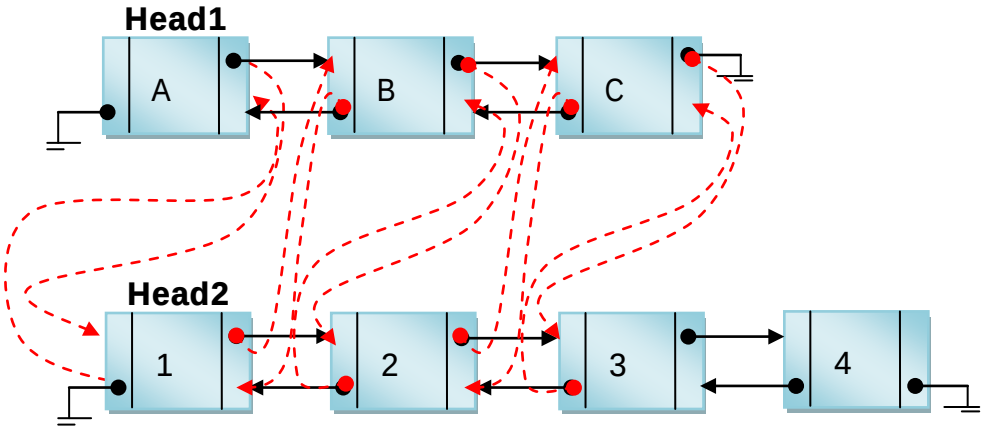
Caso 1: Las dos listas tienen el mismo tamaño.



Caso 2: El tamaño de la primera lista es mayor al tamaño de la segunda lista.



Caso 3: El tamaño de la segunda lista es mayor al tamaño de la primera lista.



Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
    p ← head1, q ← head2
    Mientras (p(sig) <> null)  y (q(sig) <> null)
        r ← q(sig)
        q(sig) ← p(sig)
        p(sig(ant)) ← q
        p(sig) ← q
        q(ant) ←p
        p ← q(sig)
        q ← r
    Si p(sig) <> null y q(sig) = null entonces
        q(sig) ← p(sig)
        p(sig(ant)) ← q
        p(sig) ← q
        q(ant) ← p
    De lo contrario
        p(sig) ← q
        q(ant) ← p
De lo contrario
    Mensaje ('No hay listas...')
```

Código en C++

```
void lista_doble_lineal::intercalar(lista_doble_lineal &a)
{
    nodo p,q,r;
    if (head != NULL && a.head != NULL)
    {
        p = head;
        q = a.head;
        while (p->sig != NULL && q->sig != NULL)
        {
            r = q->sig;
            q->sig = p->sig;
            p->sig->ant = q;
```

```
            p->sig = q;
            q->ant = p;
            p = q->sig;
            q = r;
        }
        if ((p->sig != NULL) && ( q->sig == NULL))
        {
            q->sig = p->sig;
            p->sig->ant = q;
            p->sig = q;
            q->ant = p;
        }
        else
        {
            p->sig = q;
            q->ant = p;

        }
        a.head=NULL;
    }
    else
        cout << "No hay listas...";
}
```

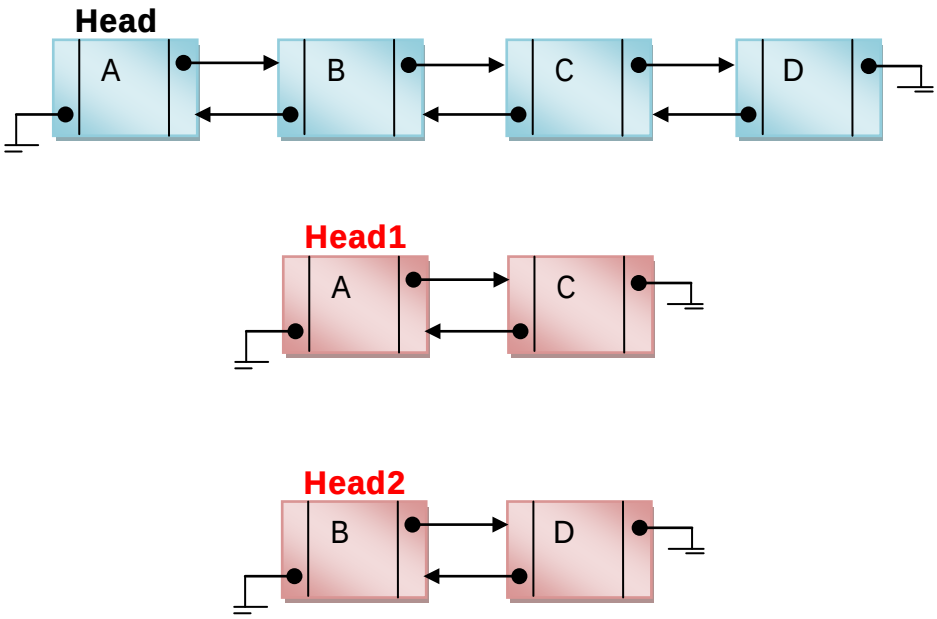
4.17 Particionar una lista

Diseñar un algoritmo que permita particionar los nodos de una lista doble lineal.

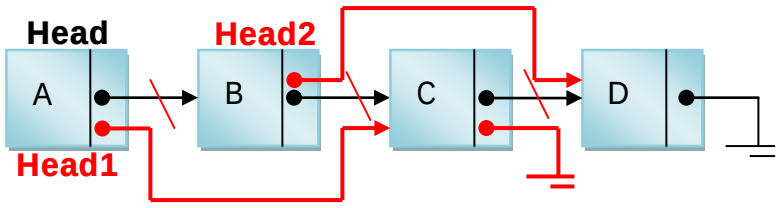
Análisis del problema

La operación de particionar una lista consiste en que tomando como base una lista doble lineal se construyen dos listas dobles lineales pasando los elementos que se encuentren en una posición impar a la primera lista y los elementos que se encuentren en una posición par se pasan a la segunda lista. La solución de este problema se puede hacer de dos maneras:

Solución 1: Crear cada uno de los nodos de las dos nuevas listas y copiar los datos que se encuentren en la lista original a la lista que le corresponda, y al final eliminar todos los nodos de la lista original.



Solución 2: En esta solución no es necesario crear y eliminar nodos, lo único que se hace es ligar todos los nodos que se encuentran en una posición impar para formar la primera de las listas y todos los nodos que se encuentran en una posición par para formar la segunda de las listas.



Solución 1

```
Si head <> null   y head(sig) <> null entonces
    p ← head
    q ← head(sig)
    Mientras p <> null
        Nuevo (r)
        r(dato) ← p(dato)
        Si head1 = null entonces
            head1 ← r
            rr ← head1
            rr(ant) ← null
        de lo contrario
            rr(sig) ← r
            r(ant) ← rr
            rr ← r
        rr(sig) ← null
        Si q<> null entonces
            Nuevo (s)
            s(dato) ← q(dato)
            Si head2 = null entonces
                head2 ← s
                ss ← head2
                ss(ant) ← null
            de lo contrario
                ss(sig) ← s
                s(ant) ← ss
                ss ← s
            ss(sig) ← null
        Si q<>null entonces
            p ← q(sig)
            Si p <> null entonces
                q ← p(sig)
            De lo contrario
                q ← null
        De lo contrario
            p = null
    p ← head
    Mientras p <> null
        head ← head(sig)
        Eliminar (p)
        p ← head
De lo contrario
    Mensaje (No hay suficientes nodos para particionar)
```

Solución 2

```
Si head <> null  y head(sig) <> null entonces
    head1 ← head
    head2 ← head(sig)
    p ← head1
    p(ant) ← null
    q ← head2
    q(ant) ← null
    Mientras q <> null
        p(sig) ← q(sig)
        Si q(sig) <> null entonces
            q(sig(ant)) ← p
        p ← q
        q ← q(sig)
    p(sig) ← null
De lo contrario
    Mensaje (No hay suficientes nodos para particionar)
```

Código en C++

```
void lista_doble_lineal::particionar(lista_doble_lineal
&a, lista_doble_lineal &b)
{
   nodo p,q,r,rr,s,ss;
   if (head != NULL && head->sig != NULL)
   {
       p = head;
       q = head->sig;
       while (p != NULL)
       {
            r = nuevo();
            r->dato = p->dato;
            if (a.head == NULL)
            {
                a.head = r;
                rr = a.head;
                rr->ant = NULL;
```

```
            }
            else
            {
                rr->sig = r;
                r->ant = rr;
                rr = r;
            }
            rr->sig = NULL;
            if (q!=NULL)
            {
                s = nuevo();
                s->dato = q->dato;
                if (b.head == NULL)
                {
                    b.head = s;
                    ss = b.head;
                    ss->ant = NULL;
                }
               else
                {
                    ss->sig= s;
                    s->ant = ss;
                    ss = s;
                }
                ss->sig = NULL;
            }
            if (q!=NULL)
            {
                p = q->sig;
                if (p != NULL)
                    q = p->sig;
                else
                    q = NULL;
            }
            else
                p = NULL;
        }
        p = head;
        while (p != NULL)
        {
            head = head->sig;
            delete (p);
            p = head;
        }
```

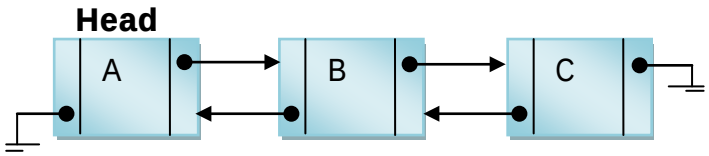
```
    }
    else
        cout << "No hay suficientes nodos...";
}
```

4.18 Buscar un elemento

Diseñar un algoritmo que permita buscar un elemento x en una lista doble lineal.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x, hasta que se encuentre el nodo con el valor de x o que se acabe la lista.



Solución en pseudocódigo

```
Si head <> null
    existe ← falso
    Leer (dato)
    p ← head
    Mientras  p<>null y existe = falso
        Si p(dato) = dato
            existe ← verdadero
        p ← p(sig)
    Si existe = verdadero entonces
        Mensaje (Si se encuentra el elemento x)
    De lo contrario
        Mensaje (No se encuentra el elemento x)
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

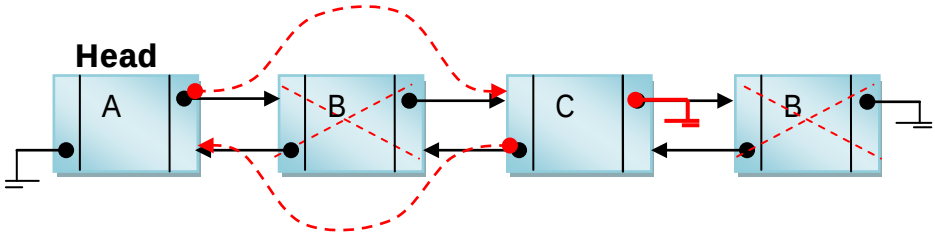
```
int lista_doble_lineal::buscar(char valor)
{
   nodo p;
   int existe;
   if (head != NULL)
   {
       existe = false;
       p=head;
       while (p!=NULL && existe == false)
       {
           if (p->dato == valor)
           {
               existe = true;
           }
           p = p->sig;
       }
       if (existe)
           cout << "Si se encuentra el elemento";
       else
           cout << "No se encuentra el elemento";
   }
   else
       cout << "Lista vacía...";
   return (existe);
}
```

4.19 Eliminar repeticiones

Diseñar un algoritmo que permita eliminar todas las repeticiones del elemento x en una lista doble lineal.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x. Si al comparar los elementos existe una coincidencia entonces se elimina el nodo, y es necesario ligar el nodo antecesor con el nodo sucesor.



Solución en pseudocódigo

```
Si head <> null  entonces
    Leer (x)
    p ← head
    Mientras  p<>null
        Si p(dato) = x entonces
            Si p = head entonces
                Head ← head(sig)
                Borrar(p)
                P ← head
            De lo contrario
                p(ant(sig)) ← p(sig)
                Si p(sig) <> null entonces
                    p(sig(ant)) ← p(ant)
                q ← p
                p ← p(sig)
                Borrar(q)
        De lo contrario
            p ← p(sig)
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

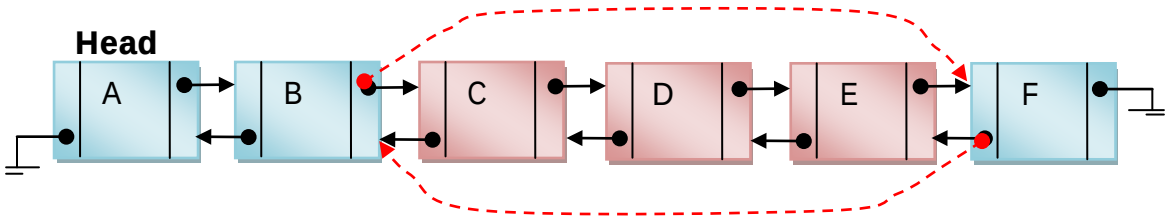
```
void lista_doble_lineal::eliminar_repetidos(char valor)
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p!= NULL)
        {
            if (p->dato == valor)
            {
                if (p==head)
                {
                    head = head->sig;
                    delete(p);
                    p = head;
                }
                else
                {
                    p->ant->sig = p->sig;
                    if (p->sig != NULL)
                        p->sig->ant = p->ant;
                    q = p;
                    p = p->sig;
                    delete(q);
                }
            }
            else
            {
                p = p->sig;
            }
        }
    }
    else
        cout << "Lista vacía...";
}
```

4.20 Eliminar subcadena

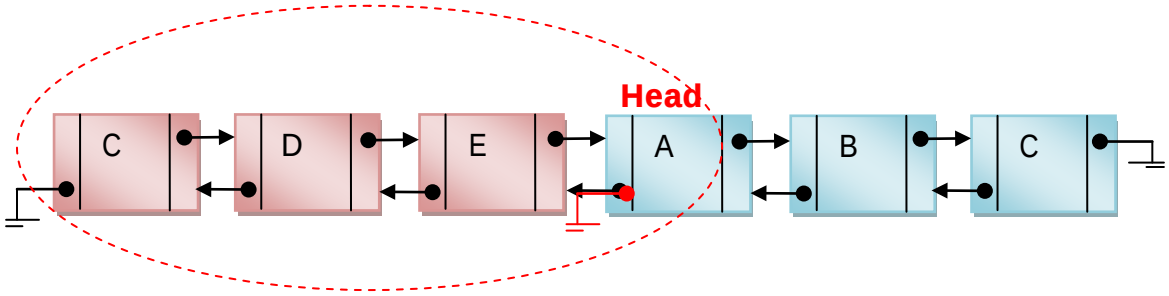
Diseñar un algoritmo que permita eliminar una subcadena de la lista doble lineal.

Análisis del problema

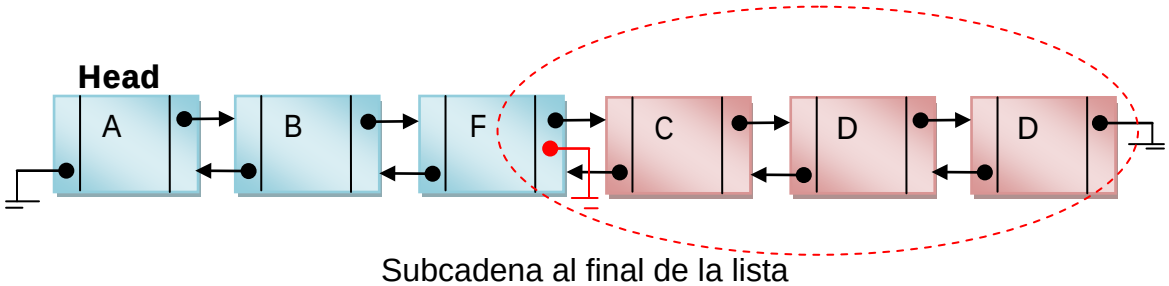
Para resolver este problema es necesario contar con el valor de la subcadena que se va a eliminar y calcular su tamaño. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación. Se localiza el inicio de la subcadena y se eliminan los nodos que forman parte de la subcadena. Es necesario validar si la subcadena está al inicio de la lista, porque en ese caso, es necesario mover la variable head al final de la subcadena.



Subcadena en una posición intermedia



Subcadena al inicio de la lista



Subcadena al final de la lista

Solución en pseudocódigo

```
Si head <> null
   Leer (subcadena)
   c ← 1, p ← head, tam ← tamaño(subcadena)
   Mientras p(sig) <> null
      p ← p(sig)
      c ← c + 1
   Si c > tam entonces
      p← head, borrado ← falso
      Mientras (p <> null) y (borrado = falso)
         Si  p(dato)  = subcadena[1] entonces
            q ← p, i←0, igual ← verdadero
            Mientras i < tam-1 y igual = verdadero y q<> null
               q ← q(sig)
               i ← i+1
               Si q <> null entonces
                  Si q(dato) <> subcadena[i] entonces
                     igual ← falso
               De lo contrario
                  igual ← falso
            Si igual = verdadero entonces
               Si p = head entonces
                     head ← q(sig)
                     Mientras p<> head
                        q ← p(sig)
                        eliminar(p)
                        p ← q
               De lo contrario
                     r ← head
                     Mientras r(sig) <> p
                        r ← r(sig)
                     r(sig) ← q(sig)
                     Mientras p<>q
                        r←p(sig)
                        eliminar(p)
                        p←r
                     Eliminar(p)
               borrado ← verdadero
         p← p(liga)
      Si borrado = falso entonces
         Mensaje ('Subcadena no existe')
   De lo contrario
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay más

Código en C++

```
void lista_doble_lineal::eliminar_subcadena(char *valor)
{
    nodo p,q,r;
    int i,c,tam,borrado,igual;
    c = 1;
    p = head;
    tam = strlen(valor);
    if (head != NULL)
    {
        while (p->sig != NULL)
        {
            p = p->sig;
            c++;
        }
        if (c >= tam)
        {
            p = head;
            borrado = false;
            while (p!= NULL && borrado == false)
            {
                if (p->dato == valor[0])
                {
                    q = p;
                    i = 0;
                    igual = true;
                    while (i < tam-1 && igual && q!= NULL)
                    {
                        q = q->sig;
                        i++;
                        if (q != NULL)
                        {
                            if (q->dato != valor[i])
                                igual = false;
                        }
                        else
                            igual = false;
                    }
                    if (igual)
                    {
                        if (p== head)
                        {
```

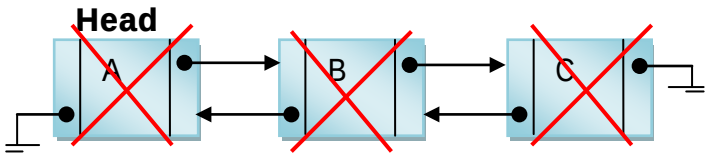
```
                                    head = q->sig;
                                    while (p!=head)
                                    {
                                        q = p->sig;
                                        delete(p);
                                        p=q;
                                    }
                                    if (c!=tam)
                                        head->ant = NULL;
                                }
                                else
                                {
                                    r = head;
                                    while (r->sig != p)
                                        r = r->sig;
                                    r->sig = q->sig;
                                    if (q->sig != NULL)
                                        q->sig->ant = r;
                                    while (p!=q)
                                    {
                                        r=p->sig;
                                        delete(p);
                                        p=r;
                                    }
                                    delete(p);
                                }
                                borrado=true;
                            }
                        }
                        if (p!=NULL)
                           p=p->sig;
                    }
                    if (borrado == false)
                        cout << "Subcadena no existe...";
                }
                else
                    cout << "Subcadena mayor que la lista...";
            }
            else
                cout << "Lista vacía";
}
```

4.21 Borrar una lista

Diseñar un algoritmo que permita borrar todos los elementos de una lista doble lineal.

Análisis del problema

Para eliminar todos los nodos de la lista se recorre toda la lista desde el primer nodo hasta llegar al valor nulo, y antes de avanzar hacia el siguiente nodo el *head* se mueve al nodo sucesor y se elimina el nodo.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p <> null
        head ← head(sig)
        Eliminar (p)
        p ← head
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_lineal::eliminar()
{
    nodo p;
    if (head != NULL)
    {
        p = head;
        while (p != NULL)
        {
            head = head->sig;
```

```
                delete(p);
                p=head;
            }
        }
        else
            cout << "Lista vacía...";
}
```

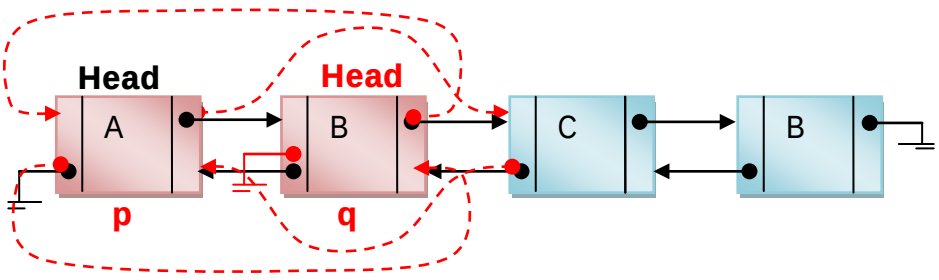
4.22 Ordenar burbuja intercambiando ligas

Diseñar un algoritmo que permita ordenar una lista doble lineal con el método de la burbuja, intercambiando las ligas.

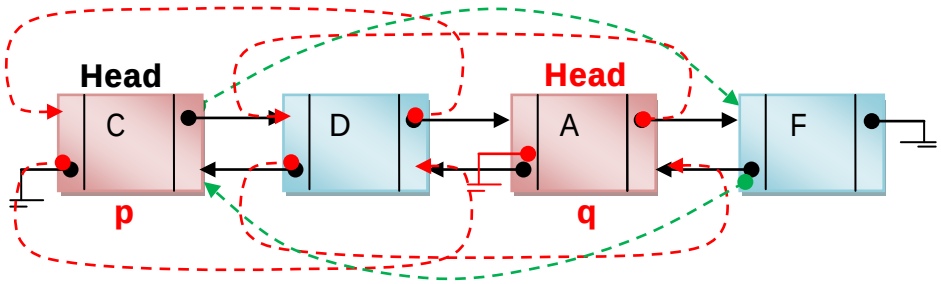
Análisis del problema

Para resolver este problema es necesario contar con al menos dos nodos. Al intercambiar las ligas los valores no cambiaran de localidad de memoria. El método de la burbuja funciona comparando el primer elemento de la lista con el resto de los elementos, en caso de que sea necesario se realiza el intercambio, posteriormente se avanza hacia el segundo elemento para compararlo con el resto y así sucesivamente hasta terminar de comparar todos los elementos. Es necesario considerar cuatro casos posibles cuando se realice un intercambio:

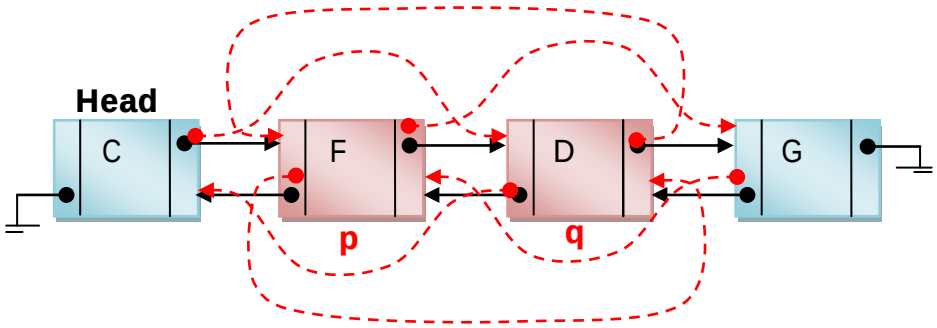
Caso 1: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.



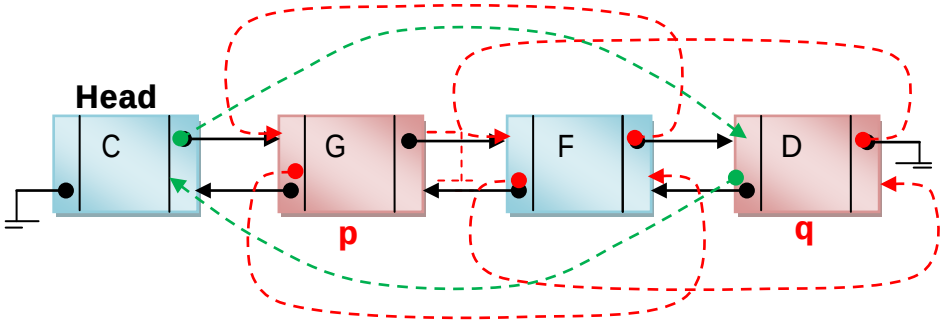
Caso 2: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Caso 3: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.



Caso 4: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras  p(sig) <>  null
        q ← p(sig)
        Mientras q <> null
            Si p(dato) > q(dato) entonces
                Si p = head y p(sig) = q entonces
                    p(sig) ← q(sig)
                    Si q(sig) <> null entonces
                        q(sig(ant) ← p
                    q(sig) ← p
                    q(ant) ← null
                    head ← q
                    p ← q
                    q ← q(sig)
                Si p = head y p(sig) <> q entonces
                    r ← q(ant)
                    r(sig) ← p
                    p(ant) ← r
                    r ← p(sig)
                    p(sig) ← q(sig)
                    Si q(sig) <> null entonces
                        q(sig(ant)) ←p
                    q(sig) ← r
                    r(ant) ← q
                    head ← q
                    q ← p
                    p ← head
                    head(ant) ← null
                Si p <> head y p(sig) = q entonces
                    p(ant(sig)) ← q
                    q(ant) ← p(ant)
                    p(sig) ← q(sig)
                    Si q(sig) <> null entonces
                        q(sig(ant)) ← p
                    q(sig) ← p
                    p(ant) ← q
                    p ← q
                    q ← q(sig)
                Si p <> head y p(sig) <> q entonces
                    p(ant(sig)) ← q
                    r ← q(ant)
                    q(ant) ← p(ant)
                    r(sig) ← p
                    p(ant) ← r
                    r ← p(sig)
                    p(sig) ← q(sig)
                    Si p(sig) <> null entonces
                        p(sig(ant)) ← p
                    q(sig) ← r
                    r(ant) ← q
                    q ← p
                    p ← r(ant)
            q ← q(sig)
        p ← p(sig)
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

```
void lista_doble_lineal::burbuja_ligas()
{
    nodo p,q,r;
    if (head != NULL)
    {
      p = head;
      while (p->sig != NULL)
      {
          q = p->sig;
          while (q != NULL)
          {
              if (p->dato > q->dato)
              {
                  if (p==head && p->sig==q)
                  {
                      p->sig = q->sig;
                      if (q->sig !=NULL)
                      {
                          q->sig->ant = p;
                      }
                      q->sig = p;
                      head = q;
                      p = q;
                      q = q->sig;
                  }
                  else
                  {
                    if (p==head && p->sig != q)
                    {
                      r = q->ant;
                      r->sig = p;
                      p->ant = r;
                      r = p->sig;
                      p->sig = q->sig;
                      if (q->sig != NULL)
                      {
                          q->sig->ant = p;
                      }
                      q->sig = r;
                      r->ant = q;
                      head = q;
```

```
                        q = p;
                        p = head;
                        head->ant = NULL;
                      }
                     else
                     {
                        if (p!=head && p->sig == q)
                         {
                           p->ant->sig = q;
                           q->ant = p->ant;
                           p->sig = q->sig;
                           if (q->sig != NULL)
                              q->sig->ant = p;
                           q->sig = p;
                           p->ant = q;
                           p = q;
                           q= q->sig;
                         }
                        else
                         {
                           if (p!=head && p->sig != q)
                            {
                              p->ant->sig = q;
                              r = q->ant;
                              q->ant = p->ant;
                              r->sig = p;
                              p->ant = r;
                              r = p->sig;
                              p->sig = q->sig;
                              if (p->sig != NULL)
                                  p->sig->ant = p;
                              q->sig = r;
                              r->ant = q;
                              q = p;
                              p = r->ant;
                            }
                         }
                      }
                   }
                q = q->sig;
             }
             p = p->sig;
          }
```

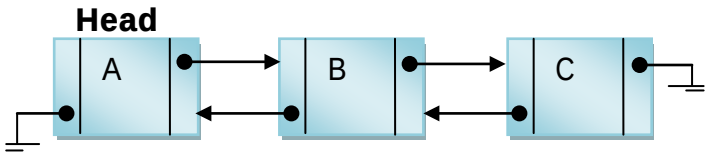
```
    }
    else
        cout << "Lista vacía...";
}
```

4.23 Buscar posición

Diseñar un algoritmo que permita devolver en qué posición se encuentra un carácter en una lista doble lineal.

Análisis del problema

Para resolver este problema es necesario contar con el valor del carácter que se buscará en la lista, recorrer toda la lista comparando el valor de cada nodo con el valor del carácter hasta encontrar el final de la lista o hasta que se encuentre el carácter.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (valor)
    p ← head
    pos ← 1
    Mientras (( p(sig) <>  null) o (p(dato)<>valor))
        p ← p(sig)
        pos ← pos + 1
    Si p(dato) = valor entonces
        Mensaje ('Posición = ', pos)
    De lo contrario
        Mensaje ('El valor no se encuentra en la lista')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

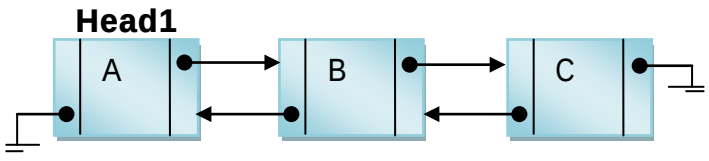
```
int lista_doble_lineal::posicion(char valor)
{
   nodo p;
   int pos;
   if (head != NULL)
   {
       p = head;
       pos = 1;
       while (p->sig != NULL && p->dato != valor)
       {
           p = p->sig;
           pos++;
       }
       if (p->dato != valor)
          pos = 0;
    }
    return pos;
}
```

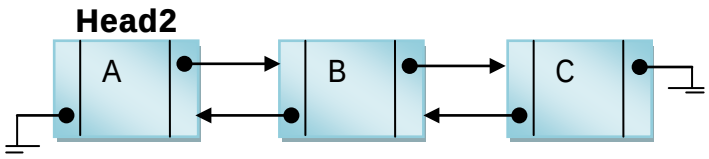
4.24 Comparar dos listas

Diseñar un algoritmo que permita comprobar si dos listas dobles lineales son exactamente iguales.

Análisis del problema

Para resolver este problema es necesario recorrer ambas listas al mismo tiempo para ir comparando nodo por nodo su valor. Antes de iniciar el recorrido se puede verificar si la longitud de ambas listas es la misma, en caso de que no sea la misma longitud se puede asumir que las listas son diferentes.





Solución en pseudocódigo

```
Si head1 <> null y head2 <> null entonces
    p ← head1
    q ← head2
    igual ← verdadero
    Mientras igual = verdadero y p <>  null y  q<> null
        Si p(dato) <> q(dato) entonces
            igual ← falso
        p ← p(sig)
        q ← q(sig)
    Si igual = verdadero entonces
        Mensaje ('Listas iguales')
    De lo contrario
        Mensaje ('Las listas no son iguales')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
int lista_doble_lineal::comparar(lista_doble_lineal a)
{
   nodo p,q;
   int igual;
   if (head != NULL && a.head != NULL)
   {
     if (tamano() == a.tamano())
      {
       p = head;
       q = a.head;
       igual = true;
```

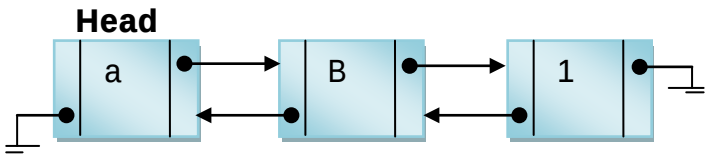
```
        while (igual && (p!= NULL && q!=NULL))
        {
            if (p->dato != q->dato)
                igual = false;
            p = p->sig;
            q = q->sig;
        }
       }
      else
         igual = false;
   }
   else
       cout << "Listas vacías...";
   return igual;
}
```

4.25 Reemplazar texto

Diseñar un algoritmo que permita reemplazar parte de una lista doble lineal por otro texto.

Análisis del problema

Para resolver este problema es necesario calcular el tamaño de la lista y el tamaño del texto que se va a reemplazar, para determinar si es posible que a partir de la posición que se indique se pueda llevar a cabo la operación. En caso de que los valores sean correctos, se ubica un apuntador en la posición en donde se hará el reemplazo y se inicia con el reemplazo.



Solución en pseudocódigo

```
Si head <> null
    Leer (pos)
    Leer (texto)
    n ← tamaño(texto)
    c ← 1
    p ← head
    Mientras p(sig) <> null
        p ← p(sig)
        c ← c + 1
    Si (pos+n) <= c entonces
        p← head
        i = 1
        Mientras i <> pos entonces
            i ← i + 1
            p ← p(sig)
        j ← 1
        Mientras j <> n
            p(dato) ← texto[j]
            p ← p(sig)
            j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_lineal::reemplazar(int pos, char *valor)
{
    nodo p;
    int n,c,i,j;
    if (head != NULL)
    {
        n = strlen(valor);
        c = 1;
        p = head;
        while (p->sig != NULL)
```

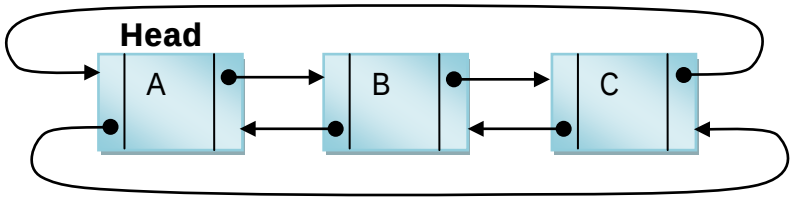
```
        {
            p = p->sig;
            c++;
        }
        if ((pos+n-1) <= c)
        {
            p = head;
            i = 1;
            while (i!=pos)
            {
                i++;
                p = p->sig;
            }
            j = 0;
            while (j!=n)
            {
                p->dato = valor[j];
                p = p->sig;
                j++;
            }
        }
        else
            cout << "Error en los valores...";
    }
    else
        cout << "Lista vacía...";
}
```

[Capítulo]

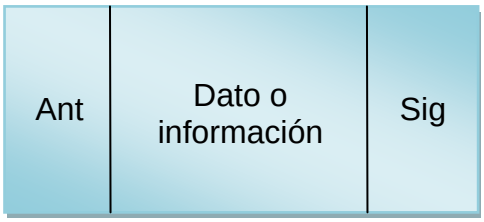
Listas Dobles Circulares

5 Listas Dobles Circulares

La característica principal de una lista doble circular es que la liga siguiente del último nodo apunta hacia el primer nodo de la lista y la liga anterior del primer nodo apunta hacia el último nodo de la lista. El valor nulo solo se utiliza cuando la lista está vacía.



El nodo de una lista doble circular debe contener como mínimo tres campos: uno para almacenar la información y otros dos para guardar la dirección de memoria del nodo antecesor y sucesor. En la figura se puede apreciar la estructura del nodo para una lista doble.



Para definir la estructura del nodo en C++ se hace lo siguiente:

```
struct apuntador
{
  char dato;
  apuntador *sig;
  apuntador *ant;
};
```

Para simplificar la asignación de memoria se utiliza la siguiente función:

```
nodo nuevo()
{
 nodo p;
 p = new struct apuntador;
 return p;
}
```

Se presenta la clase lista_doble_circular, la cual incluye la variable head y los métodos de las operaciones que se desarrollan en este capítulo para el manejo de las listas dobles circulares.

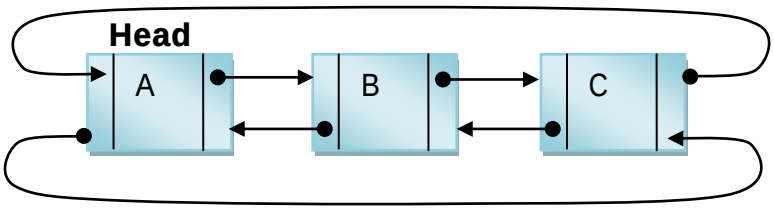
```
class lista_doble_circular
{
  nodo head;
  public:
   lista_doble_circular();
   void crear();
   void desplegar();
   void mayusculas();
   int tamano();
   void insertar_final();
   void insertar_inicio();
   void insertar(int posicion);
   void borrar_ultimo();
   void borrar_inicio();
   void borrar(int posicion);
   void desplegar_invertida();
   void burbuja();
   void invertir();
   void concatenar(lista_doble_circular &b);
   void eliminar_subcadena(int n, int x);
   void intercalar(lista_doble_circular &a);
   void particionar(lista_doble_circular &a,
        lista_doble_circular &b);
   int buscar(char valor);
   void eliminar_repetidos(char valor);
   void eliminar();
   int posicion(char valor);
   int comparar(lista_doble_circular a);
   void burbuja_ligas();
   void reemplazar(int pos, char *valor);
   void eliminar_subcadena(char *valor);
};
```

5.1 Crear una lista

Diseñar un algoritmo que permita crear una lista doble circular con n número de nodos.

Análisis del problema

Para resolver este problema es necesario la utilización de un ciclo que estará generando cada uno de los nodos que formaran parte de la lista. Es necesario introducir la información de cada uno de los nodos dentro del ciclo. Al final se liga el último nodo con el primer nodo de la lista y el primer nodo con el último nodo de la lista.



Solución en pseudocódigo

```
Si head = null entonces
    Repite
        Nuevo (p)
        Leer (p(dato))
        Si head = null entonces
            head ← p
            p(ant) ← head
            p(sig) ← head
        De lo contrario
            p(ant) ← head(ant)
            p(ant(sig)) ← p
            p(sig) ← head
            head(ant) ← p
        Leer (otro)
    Hasta (otro = NO)
De lo contrario
    Mensaje ('Lista ya contiene elementos...')
```

Código en C++

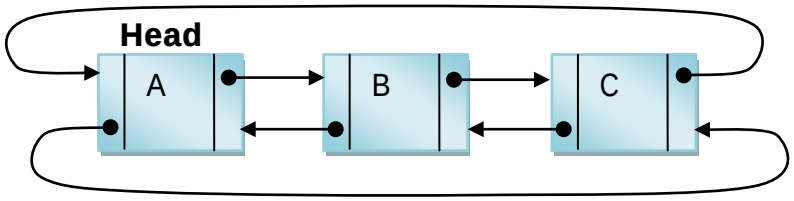
```
void lista_doble_circular::crear()
{
  nodo p;
  char otro;
  do
  {
    p=nuevo();
    cout << "p(dato) = ";
    cin >> p->dato;
    if (head == NULL)
    {
        head = p;
        p->ant = head;
        p->sig = head;
    }
    else
    {
        p->ant = head->ant;
        p->ant->sig = p;
        p->sig = head;
        head->ant = p;
    }
     cout << "Capturar otro nodo s/n ? " ;
     cin >> otro;
  } while (otro == 's');
}
```

5.2 Recorrer una lista

Diseñar un algoritmo que permita desplegar el contenido de una lista doble circular.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se recorre toda la lista desde el primer nodo donde se encuentra *head*. En una lista doble circular no existe el valor nulo, entonces para encontrar el final de la lista es necesario hacer referencia al primer nodo de la lista y tomar en cuenta esto para la condición del ciclo que recorrerá toda la lista.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Repite
        Desplegar (p(dato))
        p ← p(sig)
    Hasta (p= head)
 De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

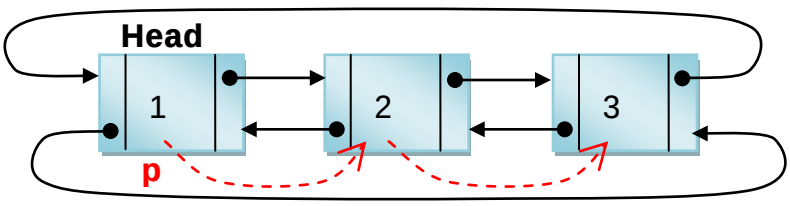
```
void lista_doble_circular::desplegar()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    do
    {
       cout << p->dato;
       p = p->sig;
    } while (p!=head);
  }
  else
    cout <<  "Lista Vacía";
}
```

5.3 Calcular tamaño

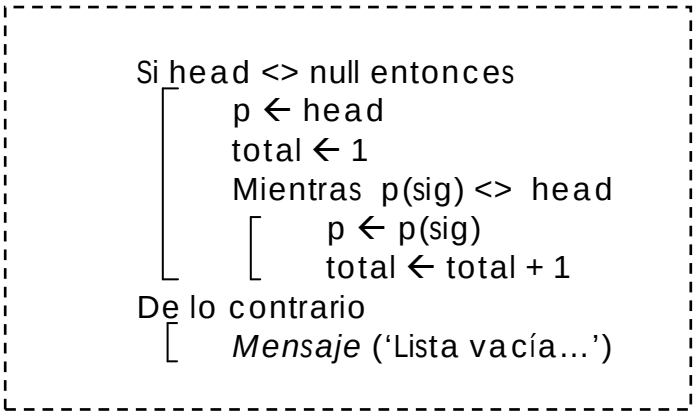
Diseñar un algoritmo que permita determinar el tamaño de una lista doble circular.

Análisis del problema

Para calcular el tamaño de la lista es necesario recorrer todos los nodos de la lista desde el primer nodo hasta encontrar el último nodo de la lista. Para contar el total de nodos se utiliza un contador que se va incrementando cada vez que se avanza una posición.



Solución en pseudocódigo



Código en C++

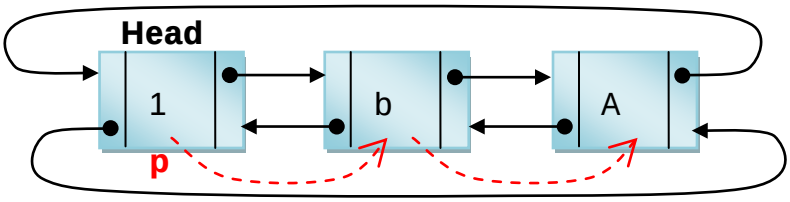
```
int lista_doble_circular::tamano()
{
  nodo p;
  int total;
  if (head != NULL)
  {
    p = head;
    total = 1;
    while (p->sig !=head)
    {
       p = p->sig;
       total++;
    }
  }
  else
    cout <<  "Lista Vacía";
  return total;
}
```

5.4 Convertir mayúsculas

Diseñar un algoritmo que permita convertir todos los elementos alfabéticos de una lista doble circular de minúsculas a mayúsculas.

Análisis del problema

Para resolver este problema es necesario recorrer toda la lista desde el inicio, para ir comparando el valor del nodo y en caso de que sea una letra convertirla a mayúscula.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Repite
        Si p(dato) es una letra entonces
            p(dato) ← mayúscula(p(dato))
        p ← p(sig)
    Hasta p=head
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_circular::mayusculas()
{
  nodo p;
  if (head != NULL)
  {
    p = head;
    do
```

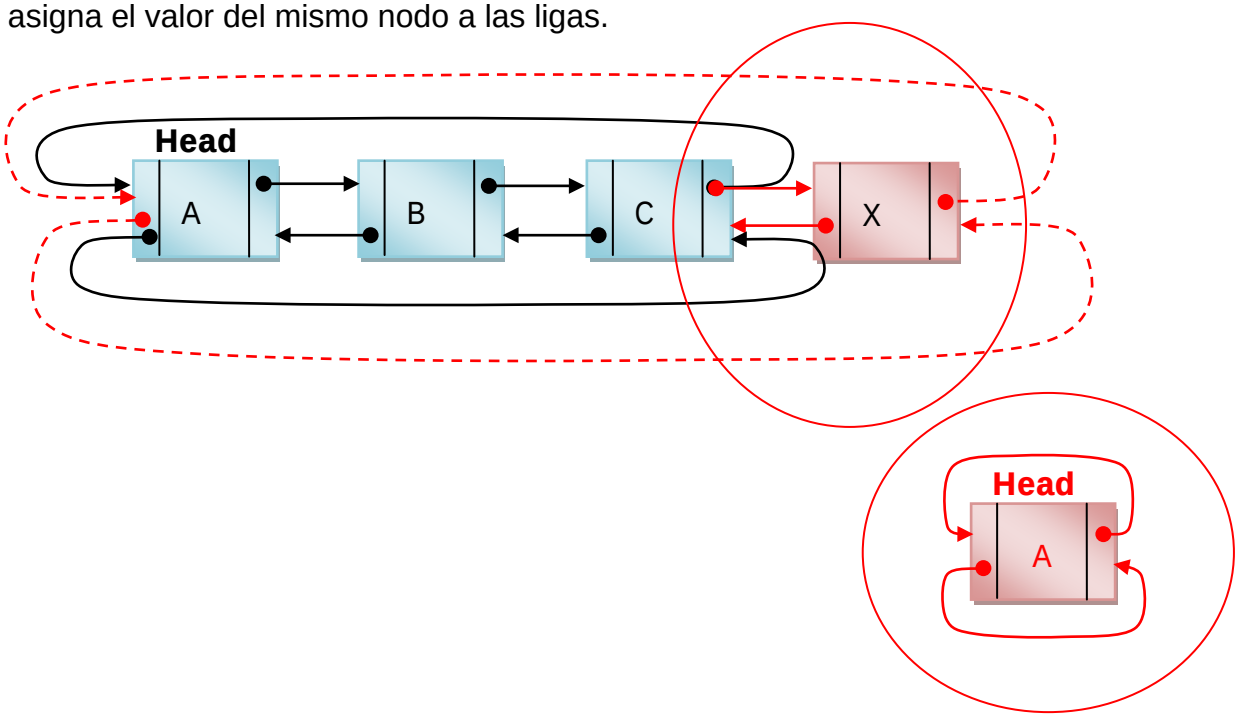
```
    {
        if (p->dato>='a' && p->dato <= 'z')
            p->dato -=32;
        p = p->sig;
    } while (p!=head);
  }
  else
    cout <<  "Lista Vacía";
}
```

5.5 Insertar al final

Diseñar un algoritmo que permita Insertar un nodo al final de una lista doble circular.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se crea el nuevo nodo y se liga el nuevo nodo con el último nodo y el primer nodo de la lista, el último nodo con el nuevo nodo y el primer nodo de la lista con el nuevo nodo. Si la lista está vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo y se les asigna el valor del mismo nodo a las ligas.



Solución en pseudocódigo

```
Nuevo(p)
Leer (p(dato))
Si head <> null entonces
    p(sig) ← head
    p(ant) ← head(ant)
    p(ant(sig)) ← p
    head(ant) ← p
De lo contrario
    head ← p
    p(sig) ← p
    p(ant) ← p
```

Código en C++

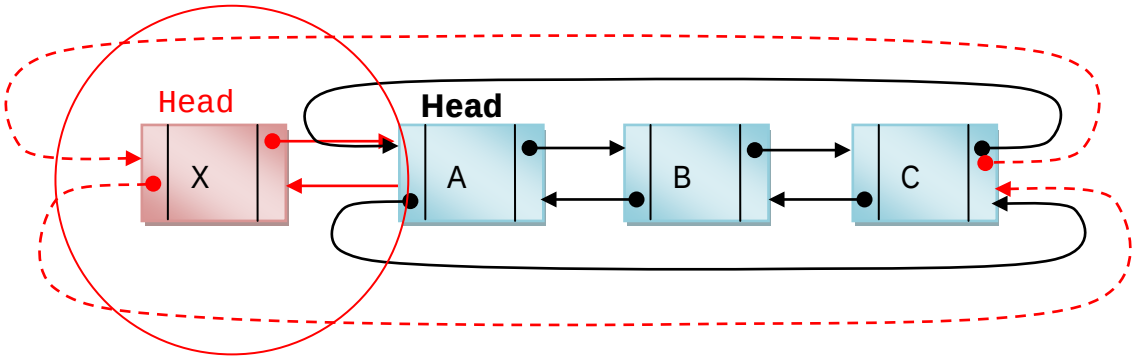
```
void lista_doble_circular::insertar_final()
{
   nodo p;
   p = nuevo();
   cout << "p(dato) = ";
   cin >> p->dato;
   if (head != NULL)
   {
       p->sig = head;
       p->ant = head->ant;
       p->ant->sig = p;
       head->ant = p;
   }
   else
   {
        head = p;
        p->sig = p;
        p->ant = p;
   }
}
```

5.6 Insertar al inicio

Diseñar un algoritmo que permita Insertar un nodo al inicio de una lista doble circular.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se crea el nuevo nodo y se liga con el primer nodo de la lista y *head* se mueve al nuevo nodo. La liga anterior del nuevo nodo se deben ligar con el último nodo de la lista y la liga siguiente del último nodo se debe ligar con el nuevo nodo. Si la lista está vacía, se crea el primer nodo de la lista ubicando a *head* en el nuevo nodo y se les asigna el valor del mismo nodo a las ligas.



Solución en pseudocódigo

```
Nuevo(p)
Leer (p(dato))
Si head = null entonces
    p(sig) ← p
    p(ant) ← p
De lo contrario
    p(sig) = head
    p(ant) = head(ant)
    head(ant(sig)) = p
    p(sig(ant)) = p
head ← p
```

Código en C++

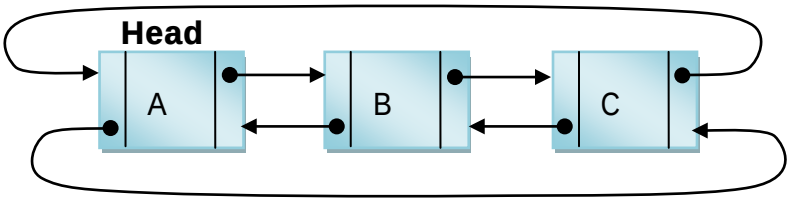
```
void lista_doble_circular::insertar_inicio()
{
   nodo p;
   p = nuevo();
   cout << "p(dato) = ";
   cin >> p->dato;
   if (head == NULL)
   {
        p->sig = p;
        p->ant = p;
   }
   else
   {
       p->sig = head;
       p->ant = head->ant;
       head->ant->sig = p;
       p->sig->ant = p;

   }
   head = p;
}
```

5.7 Insertar en cualquier posición

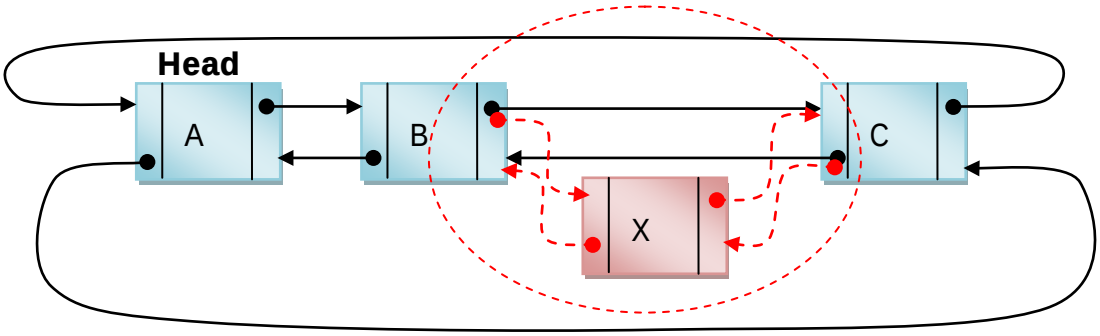
Diseñar un algoritmo que permita insertar un nodo en cualquier posición en una lista doble circular.

Análisis del problema

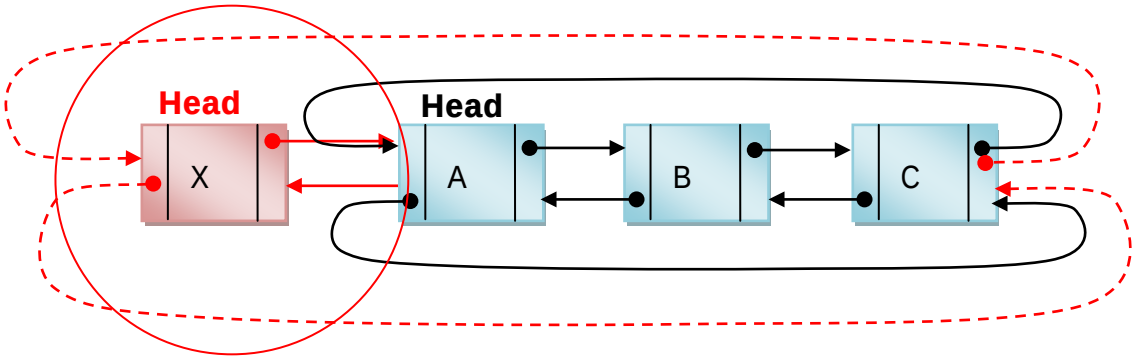
Para resolver este problema es necesario determinar si la lista contiene elementos y la posición en la que se desea insertar el nuevo nodo es válida, es menor o igual al total de los nodos de la lista. La solución debe contemplar los casos siguientes:



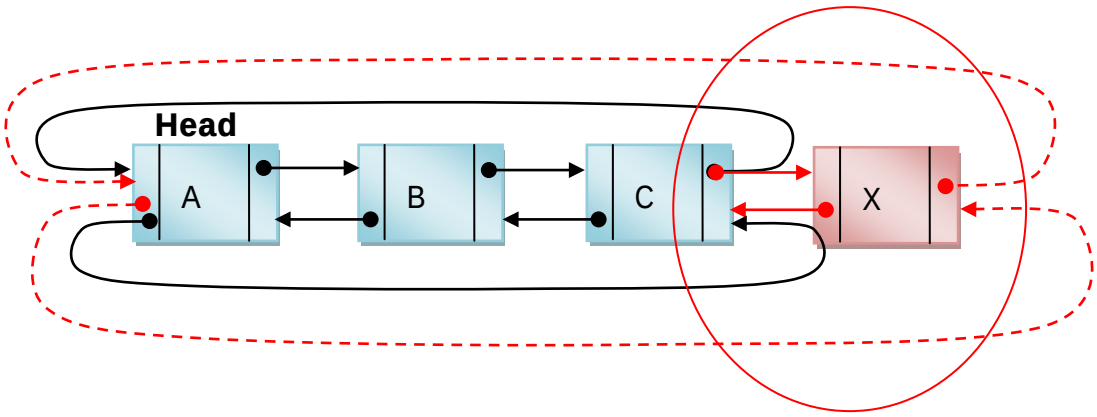
Caso 1: Insertar el nuevo nodo en una posición intermedia dentro de la lista.



Caso 2: Insertar el nuevo nodo al inicio de la lista. En este caso también se debe determinar si el nodo que se inserta es el primero de la lista, y en su caso, ubicar a head en dicho nodo y hacer que la liga siguiente del último nodo apunte hacia el nuevo nodo y la liga anterior del nuevo nodo apunte al último nodo de la lista.



Caso 3: Insertar el nuevo nodo al final de la lista y hacer que la liga del nuevo nodo apunte hacia el primer nodo de la lista donde se encuentra head.



Solución en pseudocódigo

```
Leer (posición)
p ← head
c ← 1
Mientras p(sig) <> head
    p ← p(sig)
    c ← c + 1
Si (posición  > 0) y (posición <= c+1) entonces
    Nuevo (p)
    Leer (p(dato))
    Si pos = 1 entonces
        Si head <> null entonces
            p(sig) ← head
            p(ant) ← head(ant)
            p(ant(sig)) ← p
            head(ant) ← p
            p(sig(ant)) ← p
        De lo contrario
            p(sig) ← p
            p(ant) ← p
        head ← p
    De lo contrario
        q ← head
        Para i = 1 hasta pos – 2
            q ← q(sig)
        p(sig) ← q(sig)
        p(ant) ← q
        q(sig) ← p
        p(sig(ant)) ← p
De lo contrario
    Mensaje ('Posición incorrecta...')
```

Código en C++

```
void lista_doble_circular::insertar(int posicion)
{
   nodo p,q;
   int c,i;
   p = head;
   c = 1;
```

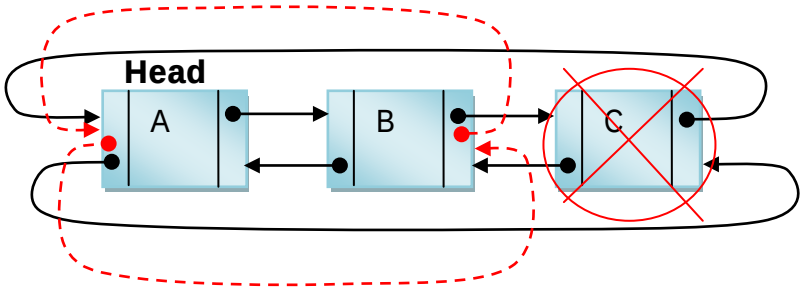
```
    while (p->sig != head)
    {
        p = p->sig;
        c++;
    }
    if ((posicion > 0) && (posicion <= c+1))
    {
        p = nuevo();
        cout << "p(dato) = ";
        cin >> p->dato;
        if (posicion==1)
        {
           if (head != NULL)
           {
               p->sig = head;
               p->ant = head->ant;
               p->ant->sig = p;
               head->ant = p;
               p->sig->ant = p;
           }
           else
           {
               p->sig = p;
               p->ant = p;
           }
           head = p;
        }
        else
        {
            q = head;
            for(i=1; i<=posicion-2; i++)
                 q = q->sig;
            p->sig = q->sig;
            p->ant = q;
            q->sig = p;
            p->sig->ant = p;
        }
    }
    else
         cout <<  "Posición Incorrecta...";
}
```

5.8 Borrar el último nodo

Diseñar un algoritmo que permita borrar el último nodo de una lista doble circular.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se posiciona un apuntador en el último nodo, para lo cual no es necesario diseñar un ciclo ya que como es una lista circular el acceso a través de la liga anterior de *head* es directo. Cuando se elimina el nodo es necesario actualizar las ligas que apuntan al último y al primer nodo de la lista. Si la lista solo contiene un nodo se elimina dicho nodo y la variable *head* se inicializa con el valor nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p← head(ant)
    Si  p = head entonces
        head ← null
    De lo contrario
        p(ant(sig)) ← head
        head(ant) ← p(ant)
    Eliminar(p)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_circular::borrar_ultimo()
{
  nodo p;
  if (head != NULL)
  {
```

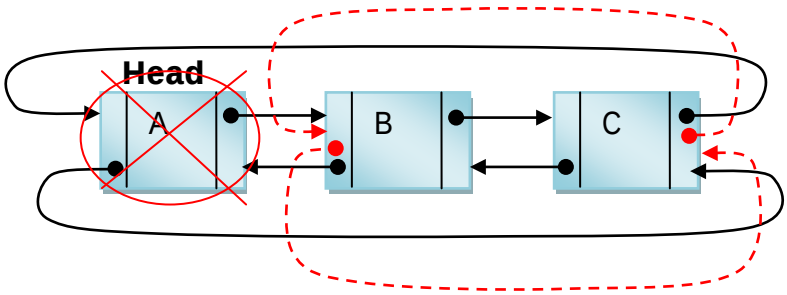
```
    p = head->ant;
    if (p == head)
    {
        head = NULL;
    }
    else
    {
        p->ant->sig = head;
        head->ant = p->ant;
    }
    delete(p);
  }
  else
    cout <<  "Lista Vacía";
}
```

5.9 Borrar el primer nodo

Diseñar un algoritmo que permita borrar el primer nodo de una lista doble circular.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos. Si la lista no está vacía se posiciona un apuntador en el primer nodo y el *head* se mueve al siguiente nodo. Cuando se elimina el nodo es necesario actualizar las ligas que apuntan al último y al primer nodo de la lista. Si la lista solo contiene un nodo se elimina dicho nodo y la variable *head* se inicializa con el valor nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p← head
    head ← head(sig)
    Si head(sig) <> head entonces
        head(ant)← p(ant)
        head(ant(sig)) ← head
    De lo contrario
        head ← null
    Eliminar(p)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

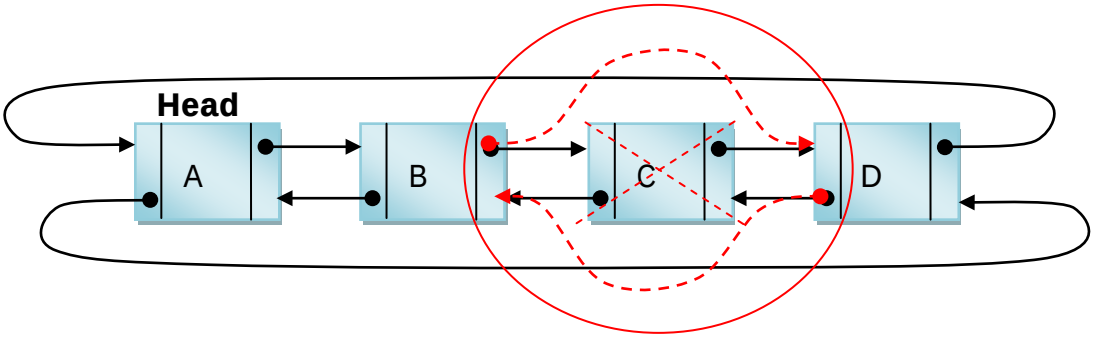
```
void lista_doble_circular::borrar_inicio()
{
  nodo p;
  if (head != NULL)
  {
    p= head;
    head = head->sig;
    if (head->sig != head)
    {
        head->ant = p->ant;
        head->ant->sig = head;
    }
    else
    {
        head = NULL;
    }
    delete(p);
  }
  else
    cout <<  "No hay elementos";
}
```

5.10 Borrar cualquier nodo

Diseñar un algoritmo que permita borrar un nodo en cualquier posición en una lista doble circular.

Análisis del problema

Para resolver este problema es necesario determinar si la lista contiene elementos y la posición del elemento que se desea eliminar es válida. En el caso de que la lista contenga un solo nodo, se elimina el nodo y se inicializa la variable head en nulo. Si la posición es la última o la primera se utilizan los algoritmos para eliminar el último o el primero nodo. Si la posición es intermedia es necesario ligar el nodo antecesor y el sucesor del nodo que se elimina.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (pos)
    p ← head
    c ← 1
    Mientras (c <> pos) y (p(sig) <> head)
        p ← p(sig)
        c ← c + 1
    Si c = pos entonces
        Si p = head entonces
            Si head(sig) <> p entonces
                head(ant(sig)) ← head(sig)
                head(sig(ant)) ← head(ant)
                head ← head (sig)
            De lo contrario
                head ← null
        De lo contrario
            p(ant(sig)) ← p(sig)
            p(sig(ant)) ← p(ant)
        Eliminar (p)
    De lo contrario
        Mensaje ('Posición inválida...')
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_circular::borrar(int posicion)
{
 nodo p;
 int c;
 if (head != NULL)
 {
   p = head;
   c = 1;
   while ((c != posicion) && (p->sig != head))
   {
       p = p->sig;
       c++;
```

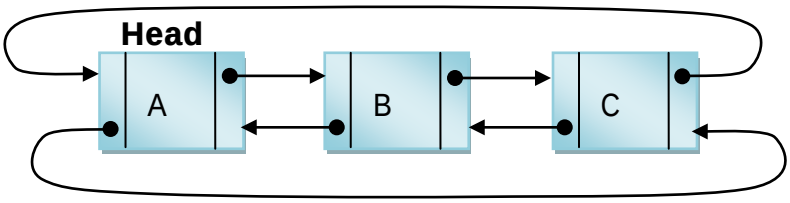
```
    }
    if (c==posicion)
    {
        if (p==head)
        {
            if (head->sig != p)
            {
                head->ant->sig = head->sig;
                head->sig->ant = head->ant;
                head = head->sig;
            }
            else
                 head = NULL;
        }
        else
        {
             p->ant->sig = p->sig;
             p->sig->ant = p->ant;
        }
        delete (p);
    }
    else
        cout <<  "Posición inválida...";
  }
  else
    cout <<  "Lista Vacía";
}
```

5.11 Desplegar invertida

Diseñar un algoritmo que permita desplegar el contenido de una lista doble circular de forma inversa.

Análisis del problema

Para resolver este problema utilizando listas dobles circulares es necesario ubicar un apuntador en el último nodo de la lista y a partir de esa posición recorrer la lista hasta llegar al primer nodo. Se tiene que verificar si la lista contiene nodos.



Solución en pseudocódigo

```
Si head <> null entonces
    p← head(ant)
    Repite
        Desplegar (p(dato))
        p ← p(ant)
    Hasta p = head
 De lo contrario
    Mensaje ('Lista vacía')
```

Código en C++

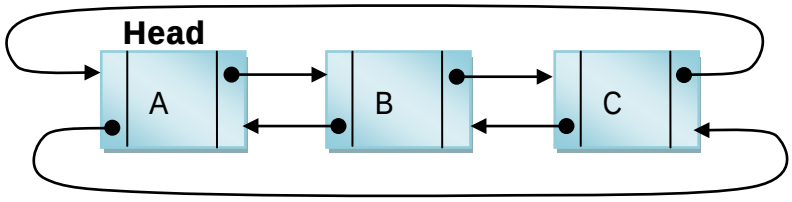
```
void lista_doble_circular::desplegar_invertida()
{
   nodo p;
   if (head != NULL)
   {
      p = head->ant;
     do
      {
          cout << p->dato;
          p = p->ant;
      }
      while (p != head);
      cout << p->dato;
   }
   else
      cout <<  "Lista Vacía";
}
```

5.12 Ordenar burbuja

Diseñar un algoritmo que permita ordenar una lista doble circular utilizando el método de la burbuja.

Análisis del problema

Para la implementación del método de ordenación de la burbuja se requieren dos ciclos anidados para ir comparando los elementos y hacer los intercambios que sean necesarios.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(sig) <> head
        q ← p(sig)
        Mientras (q <> head)
            Si (q(dato > p(dato) entonces
                aux ← p(dato)
                p(dato) ← q(dato)
                q(dato) ← aux
            q ← q(sig)
        p ← p(sig)
De lo contrario
    Mensaje ('No hay elementos...')
```

Código en C++

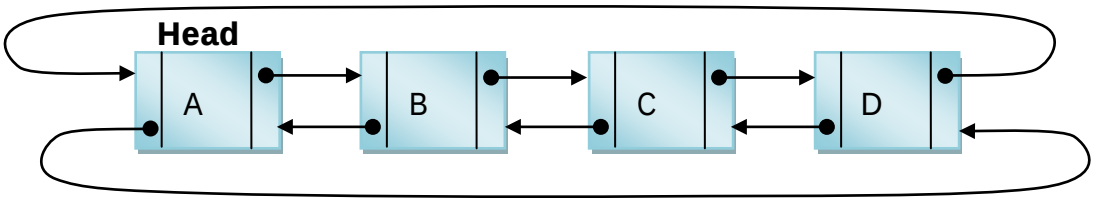
```
void lista_doble_circular::burbuja()
{
  nodo p,q;
  char aux;
  if (head != NULL)
  {
      p = head;
      while (p->sig !=head)
      {
          q = p->sig;
          while (q != head)
          {
              if (q->dato < p->dato)
              {
                  aux = p->dato;
                  p->dato = q->dato;
                  q->dato = aux;
              }
              q = q->sig;
          }
          p = p->sig;
      }
  }
  else
      cout <<  "No hay elementos";
}
```

5.13 Invertir una lista

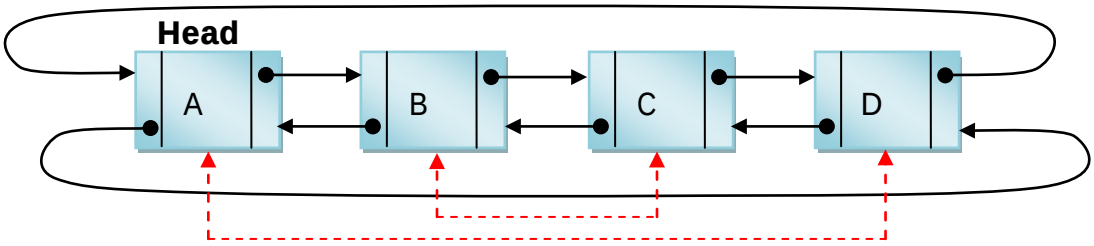
Diseñar un algoritmo que permita invertir los nodos de una lista doble circular.

Análisis del problema:

La operación de invertir una lista puede hacerse de dos maneras: la primera intercambiando únicamente los datos y la segunda dejando los datos en la posición de memoria en la que se encuentran y cambiar el ligado de los nodos para que la lista quede invertida.



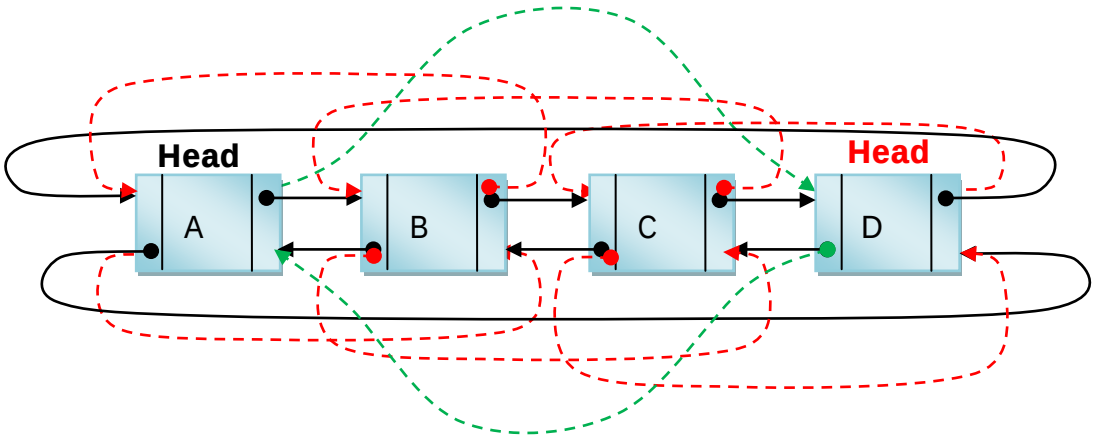
a) Moviendo datos



Solución en pseudocódigo

```
Si head(sig) <> head entonces
    p ← head
    Mientras p(sig) <> head
        p ← p(sig)
    head2 ← p
    Mientras (p <> head)
        q ← p(ant)
        aux ← p(sig)
        p(sig) ← p(ant)
        p(ant) ← aux
        p ← q
    q(ant)←p
    q(sig)←head2
    head ← head2
De lo contrario
    Mensaje ('No hay elementos...')
```

b) Moviendo ligas



Código en C++

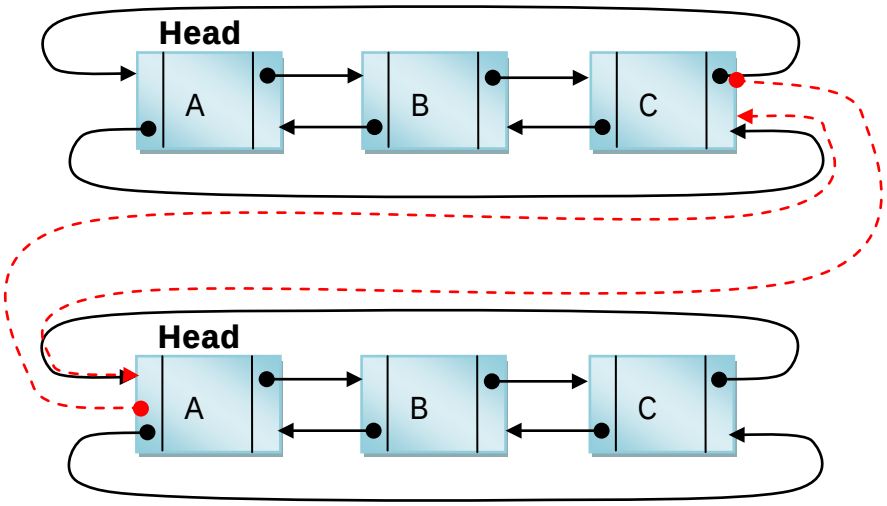
```
void lista_doble_circular::invertir()
{
   nodo p,q,r,aux;
   if ((head != NULL) && (head->sig != head))
   {
       p = head->ant;
       r = p;
       while (p != head)
       {
           q = p->ant;
           aux = p->sig;
           p->sig = p->ant;
           p->ant = aux;
           p = q;
       }
       q->ant = p;
       q->sig = r;
       head = r;
   }
   else
       cout <<  "No hay suficientes elementos";
}
```

5.14 Concatenar dos listas

Diseñar un algoritmo que permita concatenar los nodos de dos listas dobles circulares.

Análisis del problema

Para resolver este problema es necesario verificar que las dos listas contengan elementos. Se recorre la primera lista hasta el último nodo y se liga con el primer nodo de la segunda lista y la liga anterior del primer nodo de la segunda lista se liga con el último nodo de la primera lista.



Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
    p ← head1(ant)
    p(sig) ← head2
    q ← head2(ant)
    head2(ant) ← p
    head ← head1
    q(sig) ← head
    head(ant) ← q
    head2 ← null
De lo contrario
    Mensaje ('No hay listas...')
```

Código en C++

```
void lista_doble_circular::concatenar(lista_doble_circular &b)
{
   nodo p,q;
   if (head != NULL && b.head != NULL)
   {
       p = head->ant;
       p->sig = b.head;
       q = b.head->ant;
       b.head->ant = p;
```

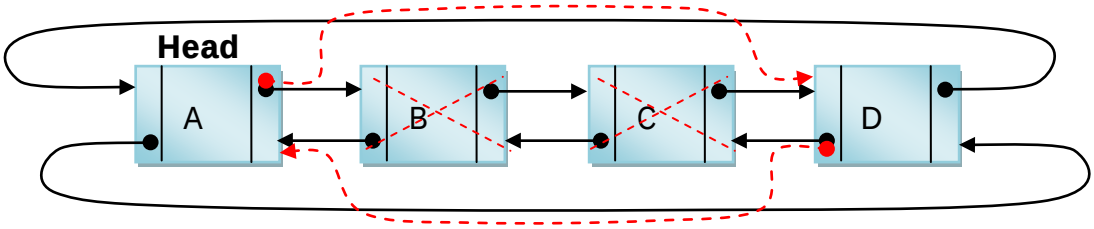
```
        q->sig = head;
        head->ant = q;
        b.head = NULL;
    }
    else
        cout <<  "No hay listas";
}
```

5.15 Eliminar n número de nodos

Diseñar un algoritmo que permita eliminar n número de nodos en una lista doble circular a partir de la posición x.

Análisis del problema

Para resolver este problema es necesario contar con el valor de n y x. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación, es decir, si existe la posición a partir de la cual se van a eliminar elementos y suficientes nodos para cumplir con el valor de n.



Solución en pseudocódigo

```
Si head <> null
    Leer (x,n)
    c ← 1
    p ← head
    Mientras p(sig) <> head
            p ← p(sig)
            c ← c + 1
    Si (x+n-1) <= c entonces
        p← head
        Si  x = 1 entonces
            j ← 0
            Mientras j <> n
                head ← head(sig)
                head(ant) ← p(ant)
                Eliminar(p)
                p ← head
                j ← j + 1
            Si n = c entonces
                head ← null
            De lo contrario
                head(ant(sig)) ← head
        De lo contrario
            i← 2
            Mientras i <> x entonces
                i ← i + 1
                p ← p(sig)
            j ← 1
            Mientras j <> n
                q ← p(sig)
                p(sig) ← q(sig)
                q(sig(ant)) ← p
                Eliminar(q)
                j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores')

De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_circular::eliminar_subcadena(int n, int x)
{
    nodo p,q;
    int c,i,j;
    if (head != NULL)
    {
        c = 1;
        p = head;
        while (p->sig != head)
        {
            p = p->sig;
            c++;
        }
        if ((x+n-1)<=c)
        {
            p = head;
            if (x==1)
            {
                j = 0;
                while (j!= n)
                {
                   head = head->sig;
                   head->ant = p->ant;
                   delete(p);
                   p=head;
                   j++;
                }
                if (n==c)
                    head = NULL;
                else
                    head->ant->sig = head;
            }
           else
            {
                i = 2;
                while (i!=x)
                {
                   i++;
                   p = p->sig;
                }
                j = 0;
                while (j!=n)
```

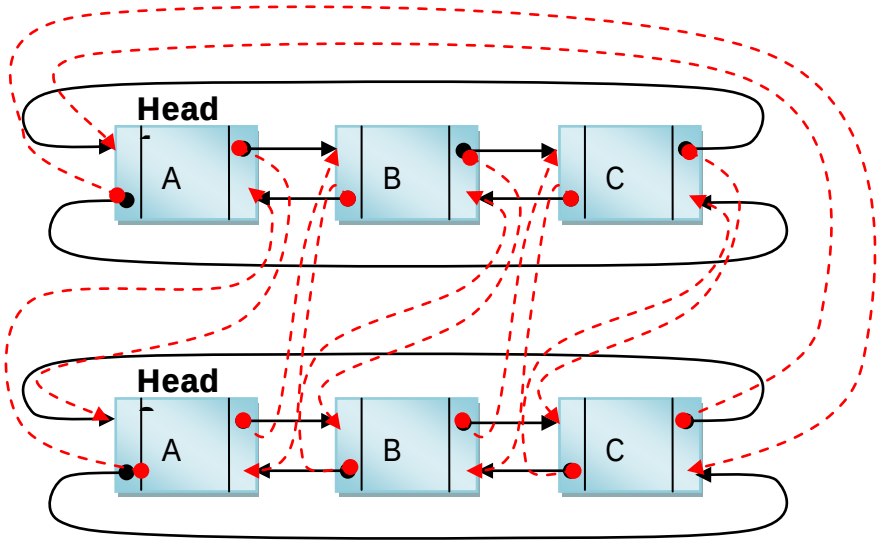
```
                    {
                        q = p->sig;
                        p->sig = q->sig;
                        q->sig->ant = p;
                        delete(q);
                        j++;
                    }
                }
            }
            else
              cout << "Error en los valores";
        }
        else
            cout << "Lista vacía";
}
```

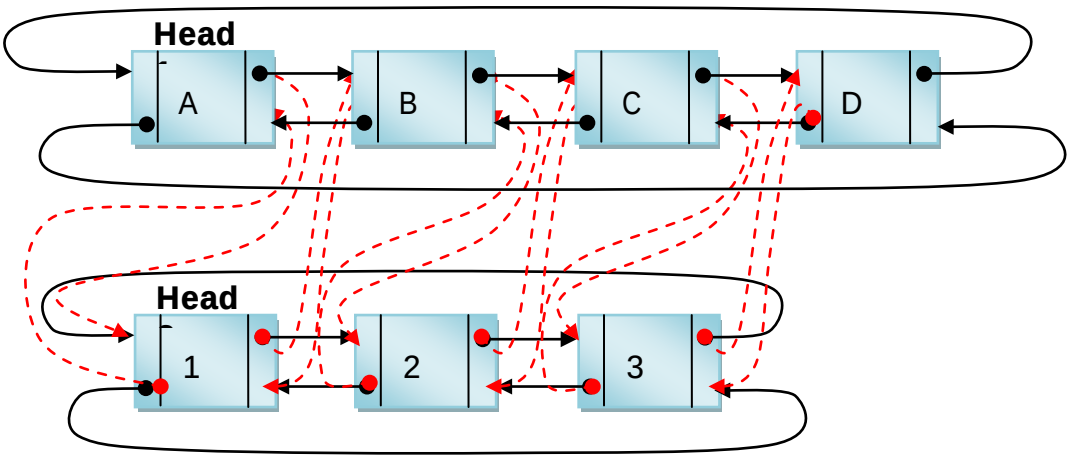
5.16 Intercalar dos listas

Diseñar un algoritmo que permita Intercalar los nodos de dos listas dobles circulares.

Análisis del problema

La operación de intercalar dos listas circulares se lleva a cabo recorriendo ambas listas al mismo tiempo. Es necesario verificar si las dos listas contienen elementos y comparar el tamaño de cada una de las listas. En el caso de que las listas sean del mismo tamaño, el último nodo de la segunda lista se encadena con el primer nodo de la primera de las listas.





Solución en pseudocódigo

```
Si head1 <> null  y head2 <> null entonces
    p ← head1, q ← head2
    Mientras (p(sig) <> head1)  y (q(sig) <> head2)
        r ← q(sig)
        q(sig) ← p(sig)
        p(sig(ant)) ← q
        p(sig) ← q
        q(ant) ←p
        p ← q(sig)
        q ← r
    Si p(sig) <> head  y q(sig) = head entonces
        q(sig) ← p(sig)
        p(sig(ant)) ← q
        p(sig) ← q
        q(ant) ← p
    De lo contrario
        Si p(sig) = head y q(sig) <> head entonces
            p(sig) ← q
            q(ant) ← p
            Mientras q(sig) <> head
                q ← q(sig)
        De lo contrario
            p(sig) ← q
            q(ant) ← p
        q(sig) ← head1
        head1(ant) ← q
De lo contrario
    Mensaje (No hay listas)
```

Código en C++

```
void lista_doble_circular::intercalar(lista_doble_circular &a)
{
    nodo p,q,r;
    if (head != NULL && a.head != NULL)
    {
        p = head;
        q = a.head;
        while (p->sig != head && q->sig != a.head)
        {
            r = q->sig;
            q->sig = p->sig;
            p->sig->ant = q;
            p->sig = q;
            q->ant = p;
            p = q->sig;
            q = r;
        }
        if (p->sig != head  && q->sig == a.head)
        {
            q->sig = p->sig;
            p->sig->ant = q;
            p->sig = q;
            q->ant = p;
        }
        else
        {
            if (p->sig == head && q->sig != a.head)
            {
                p->sig = q;
                q->ant = p;
                while (q->sig != a.head)
                    q = q->sig;
            }
            else
            {
                p->sig = q;
                q->ant = p;
            }
            q->sig = head;
            head->ant = q;
```

```
        }
        a.head=NULL;
    }
    else
        cout << "No hay listas...";
}
```

5.17 Particionar una lista

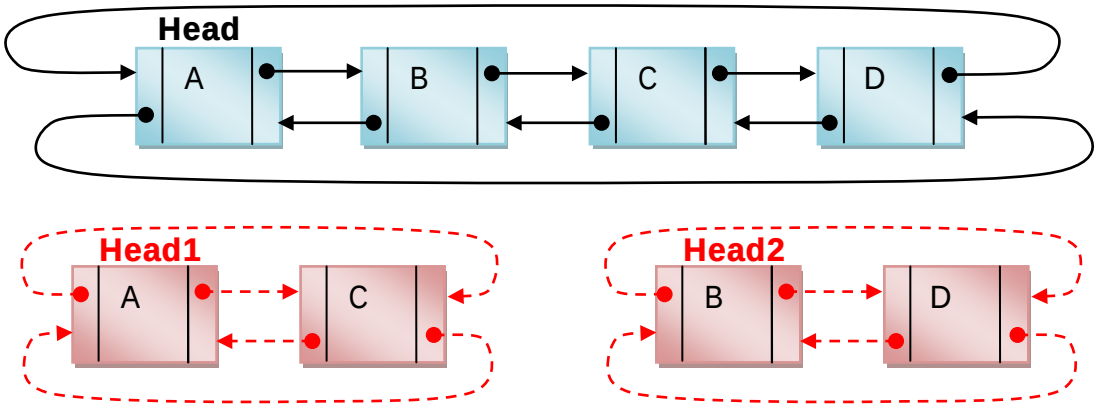
Diseñar un algoritmo que permita particionar los nodos de una lista doble circular.

Análisis del problema

La operación de particionar una lista consiste en que tomando como base una lista doble circular se construyen dos listas circulares dobles pasando los elementos que se encuentren en una posición impar a la primera lista y los elementos que se encuentren en una posición par se pasan a la segunda lista. La solución de este problema se puede hacer de dos maneras:

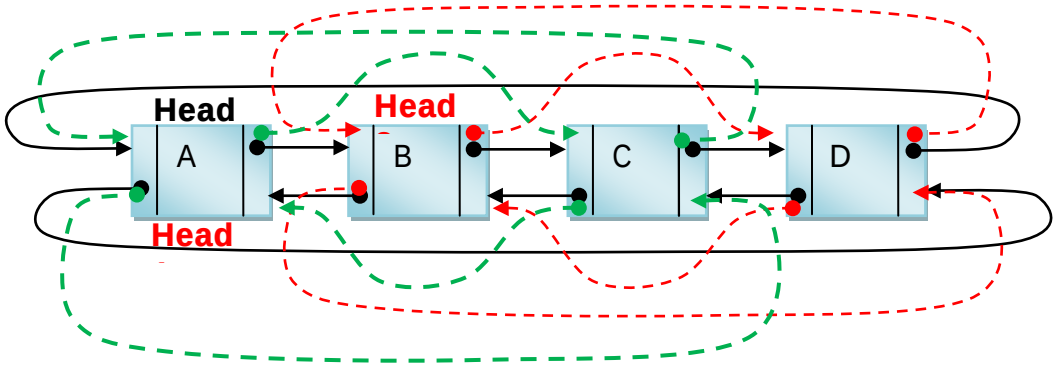
Solución 1:

Crear cada uno de los nodos de las dos nuevas listas y copiar los datos que se encuentren en la lista original a la lista que le corresponda, y al final eliminar todos los nodos de la lista original.



Solución 2:

En esta solución no es necesario crear y eliminar nodos, lo único que se hace es ligar todos los nodos que se encuentran en una posición impar para formar la primera de las listas y todos los nodos que se encuentran en una posición par para formar la segunda de las listas.



Solución en pseudocódigo

```
Si head <> null  y head(sig) <> head entonces
    head1 ← head
    head2 ← head(sig)
    p ← head1
    q ← head2
    i ← 0
    Mientras q(sig) <> head
        p(sig) ← q(sig)
        q(sig(ant) ← p
        p ← q
        q ← q(sig)
        i ← i +1
    Si (i%2 = 0) entonces
        p(sig) ← head1
        head1(ant) ← p
        q(sig) ← head2
        head2(ant)  ←  q
    De lo contrario
        p(sig) ← head2
        head2(ant) ← p
        q(sig) ← head1
        head1(ant) ← q
De lo contrario
    Mensaje (No hay suficientes nodos para particionar)
```

Código en C++

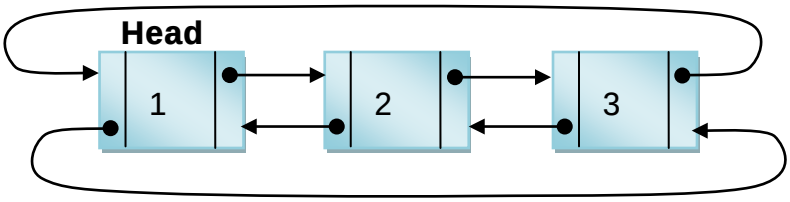
```
void lista_doble_circular::particionar(lista_doble_circular &a, lista_doble_circular &b)
{
   nodo p,q;
   int i;
   i = 0;
   if (head != NULL && head->sig != head)
   {
       p = head;
       q = head->sig;
       a.head = p;
       b.head = q;
       while (q->sig != head)
       {
            p->sig = q->sig;
            q->sig->ant = p;
            p = q;
            q = q->sig;
            i++;
       }
       if (i%2 == 0)
       {
            p->sig = a.head;
            a.head->ant = p;
            q->sig = b.head;
            b.head->ant = q;
       }
       else
       {
            p->sig = b.head;
            b.head->ant = p;
            q->sig = a.head;
            a.head->ant = q;
       }
   }
   else
       cout << "No hay suficientes nodos...";
}
```

5.18 Buscar un elemento

Diseñar un algoritmo que permita buscar un elemento x en una lista doble circular.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x, hasta que se encuentre el nodo con el valor de x o que se acabe la lista.



Solución en pseudocódigo

```
Si head <> null
    existe ← falso
    Leer (dato)
    p ← head
    Repite
        Si p(dato) = dato
            existe ← verdadero
        p ← p(sig)
    Hasta p=head o existe = verdadero
    Si existe = verdadero entonces
        Mensaje (Si se encuentra el elemento x)
    De lo contrario
        Mensaje (No se encuentra el elemento x)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

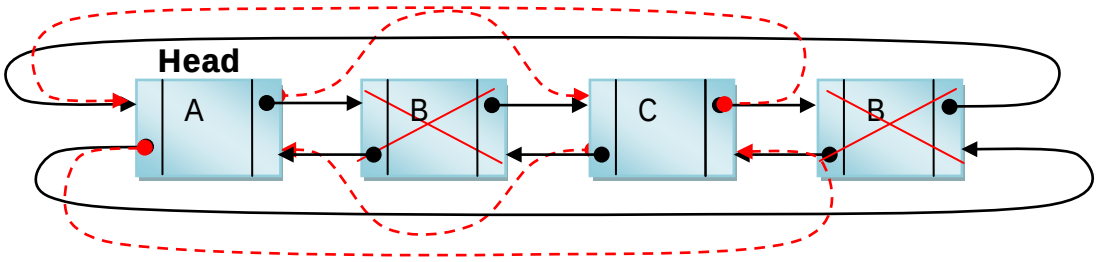
```
int lista_doble_circular::buscar(char valor)
{
   nodo p;
   int existe;
   if (head != NULL)
   {
       existe = false;
       p=head;
       do
       {
           if (p->dato == valor)
           {
               existe = true;
           }
           p = p->sig;
       } while (p!= head);
       if (existe)
           cout << "Si se encuentra el elemento";
       else
           cout << "No se encuentra el elemento";
   }
   else
       cout << "Lista vacía...";
   return (existe);
}
```

5.19 Eliminar repeticiones

Diseñar un algoritmo que permita eliminar todas las repeticiones del elemento x en una lista doble circular.

Análisis del problema

Para resolver este problema es necesario contar con el valor del elemento x, recorrer toda la lista desde el primer nodo y comparar el valor de cada nodo que se va recorriendo con el valor del elemento x. Si al comparar los elementos existe una coincidencia entonces se elimina el nodo, y es necesario ligar el nodo antecesor con el nodo sucesor.



Solución en pseudocódigo

```
Si head <> null  entonces
    Leer (x)
    p ← head
    Mientras  p(sig)<>head
        Si p(dato) = x entonces
            Si p = head entonces
                head ← head(sig)
                p(ant(sig)) ← head
                head(ant) ← p(ant)
                Eliminar(p)
                p ← head
            De lo contrario
                p(ant(sig)) ← p(sig)
                p(sig(ant)) ← p(ant)
                q ← p
                p ← p(sig)
                Eliminar(q)
        De lo contrario
            p ← p(sig)
    Si p(dato) = x entonces
        Si p = head entonces
            p(ant(sig)) ← p(sig)
            p(sig(ant)) ← p(ant)
            Eliminar(p)
        De lo contrario
            head ← null
            Eliminar(p);
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_circular::eliminar_repetidos(char valor)
{
    nodo p,q,r;
    if (head != NULL)
    {
        p = head;
       while (p->sig != head)
        {
            if (p->dato == valor)
            {
                if (p==head)
                {
                    head = head->sig;
                    p->ant->sig = head;
                    head->ant = p->ant;
                    delete(p);
                    p = head;
                }
                else
                {
                    p->ant->sig = p->sig;
                    p->sig->ant = p->ant;
                    q = p;
                    p = p->sig;
                    delete(q);
                }
            }
            else
            {
                p = p->sig;
            }
        }
     if (p->dato == valor)
       {
            if (p!= head)
            {
                p->ant->sig = p->sig;
                p->sig->ant = p->ant;
                delete(p);
            }
```

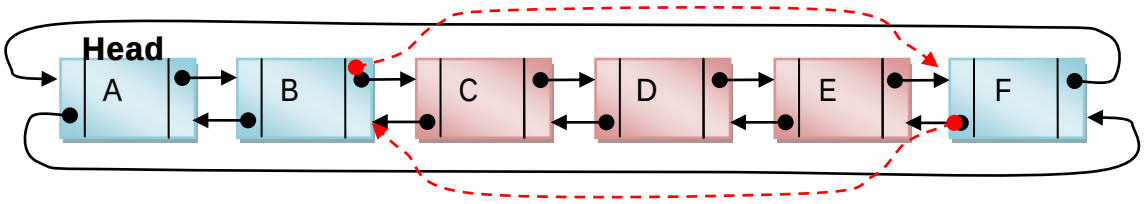
```
            else
            {
                head = NULL;
                delete(p);
            }
        }
    }
    else
        cout << "Lista vacía...";
}
```

5.20 Eliminar una subcadena

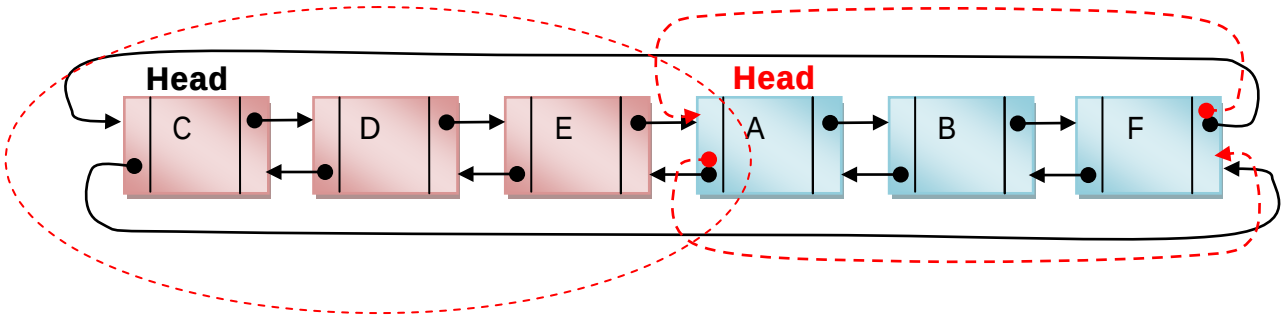
Diseñar un algoritmo que permita eliminar una subcadena de la lista doble circular.

Análisis del problema

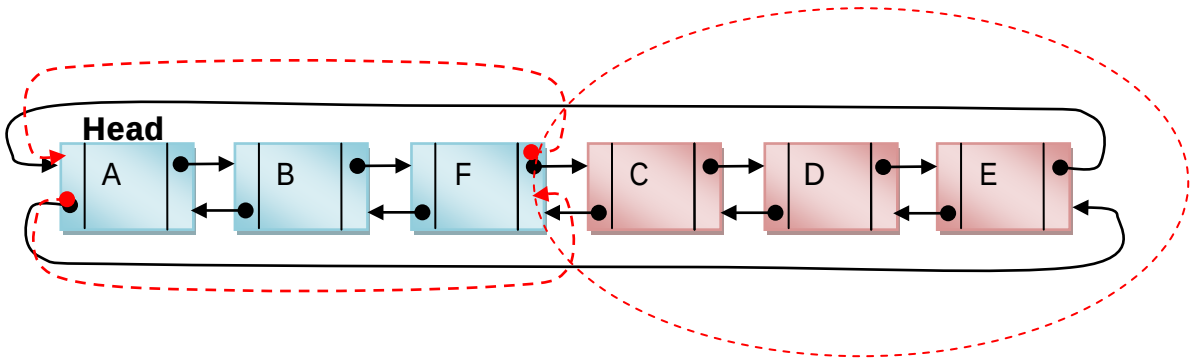
Para resolver este problema es necesario contar con el valor de la subcadena que se va a eliminar y calcular su tamaño. Se tiene que verificar si la lista contiene suficientes elementos para llevar a cabo la operación. Se localiza el inicio de la subcadena y se eliminan los nodos que forman parte de la subcadena. Es necesario validar si la subcadena está al inicio de la lista, porque en ese caso, es necesario mover la variable head al final de la subcadena.



Subcadena en una posición intermedia



Subcadena al inicio de la lista



Subcadena al final de la lista

Solución en pseudocódigo

```
Si head <> null
   Leer (subcadena)
   c ← 1, p ← head, tam ← tamaño(subcadena)
   Mientras p(sig) <> head
      p ← p(sig)
      c ← c + 1
   Si c > tam entonces
      p← head, borrado ← falso
      Mientras (p(sig) <> head y (borrado = falso)
         Si p(dato) = subcadena[1] entonces
            q ← p, i←1, igual ← verdadero
            Mientras i < tam y igual = verdadero y q<> head
               q ← q(sig)
               i ← i+1
               Si q <> head entonces
                  Si q(dato) <> subcadena[i] entonces
                     igual ← falso
               de lo contrario
                  igual ← falso
            Si igual = verdadero entonces
               Si p = head entonces
                     head ← q(sig)
                     head(ant) ← p(ant)
                     mientras p<> head
                        q ← p(sig)
                        eliminar(p)
                        p ← q
```

Valida los casos:

- ☑ Si no hay elementos
- ☑ Si hay un solo elemento
- ☑ Si hay más de un elemento

```
                                    de lo contrario
                                        r ← p(ant)
                                        r(sig) ← q(sig)
                                        q(sig(ant)) ← r
                                        mientras p<>q
                                            r←p(sig)
                                            eliminar(p)
                                            p←r
                                        Eliminar(p)
                                    borrado ← verdadero
                            p← p(sig)
                        Si borrado = falso entonces
                            Mensaje ('Subcadena no existe')
                    De lo contrario
                        Mensaje ('Subcadena mayor que la lista')
            De lo contrario
                Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_doble_circular::eliminar_subcadena(char *valor)
{
    nodo p,q,r,s;
    int i,c,tam,borrado,igual;
    c = 1;
    p = head;
    tam = strlen(valor);
    if (head != NULL)
    {
        while (p->sig != head)
        {
            p = p->sig;
            c++;
        }
        s=p;
        if (c >= tam)
        {
            p = head;
            borrado = false;
            while (p->sig!= head && borrado == false)
            {
```

```
                if (p->dato == valor[0])
                {
                    q = p;
                    i = 0;
                    igual = true;
                    while (i < tam-1 && igual && q!= s)
                    {
                        q = q->sig;
                        i++;
                        if (q != head)
                        {
                            if (q->dato != valor[i])
                                igual = false;
                        }
                        else
                            igual = false;
                    }
                if (igual)
                    {
                        if (p== head)
                        {
                            if (q->sig != head)
                            {
                                head = q->sig;
                                head->ant = p->ant;
                                head->ant->sig = head;
                                while (p!=head)
                                {
                                    q = p->sig;
                                    delete(p);
                                    p=q;
                                }
                            }
                            else
                            {
                                while (p!=q)
                                {
                                    head = head->sig;
                                    head->ant = q;
                                    q->sig = head;
                                    delete (p);
                                    p = head;
                                }
                                delete (p);
```

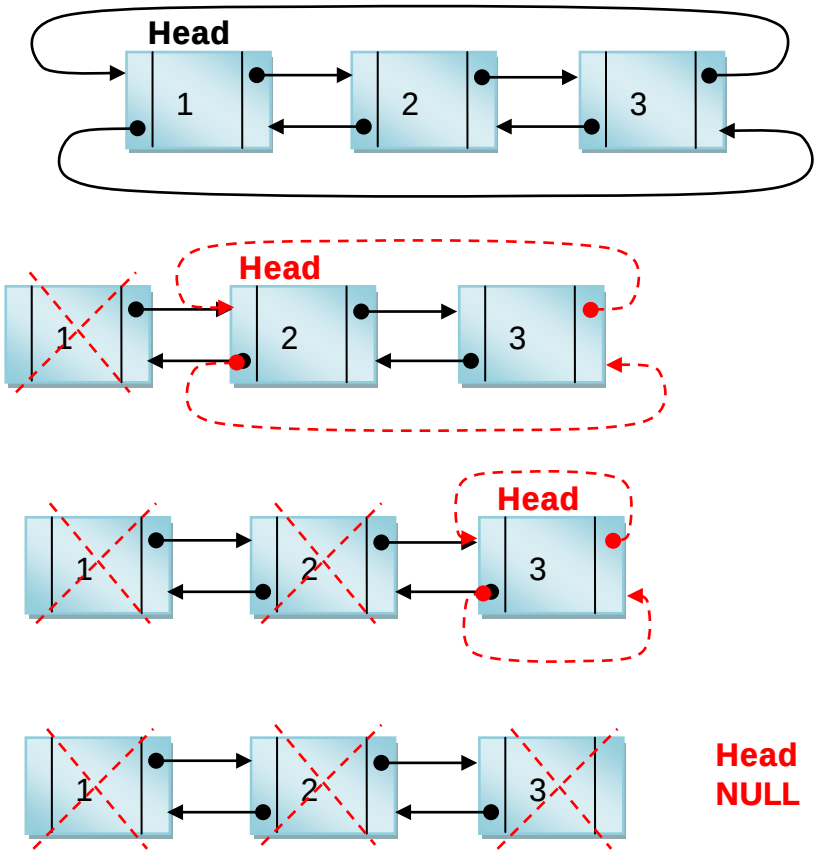
```
                                    head = NULL;
                                }
                            }
                            else
                            {
                                r = p->ant;
                                r->sig = q->sig;
                                q->sig->ant = r;
                                while (p!=q)
                                {
                                    r=p->sig;
                                    delete(p);
                                    p=r;
                                }
                                delete(p);
                            }
                        borrado=true;
                        }
                    }
                    p=p->sig;
                }
                if (borrado == false)
                    cout << "Subcadena no existe...";
            }
            else
                cout << "Subcadena mayor que la lista...";
        }
        else
            cout << "Lista vacía";
}
```

5.21 Borrar una lista

Diseñar un algoritmo que permita borrar todos los elementos de una lista doble circular.

Análisis del problema

Para eliminar todos los nodos de la lista se recorre toda la lista desde el primer nodo hasta llegar al valor nulo, y antes de avanzar hacia el siguiente nodo el *head* se mueve al nodo sucesor y se elimina el nodo.



Solución en pseudocódigo

```
Si head <> null entonces
    q← head(ant)
    Mientras  head(sig) <> head
        head ← head(sig)
        q(sig) ← head
        head(ant) ← q
        Eliminar (p)
        p ← head
    head ← null
    Eliminar (p)
De lo contrario
    Mensaje ('Lista vacía...)
```

Código en C++

```
void lista_doble_circular::eliminar()
{
    nodo p,q;
    if (head != NULL)
    {
        q = head->ant;
        while (head->sig != head)
        {
            head = head->sig;
            q->sig = head;
            head->ant = q;
            delete(p);
            p=head;
        }
        head = NULL;
        delete(p);
    }
    else
        cout << "Lista vacía...";
}
```

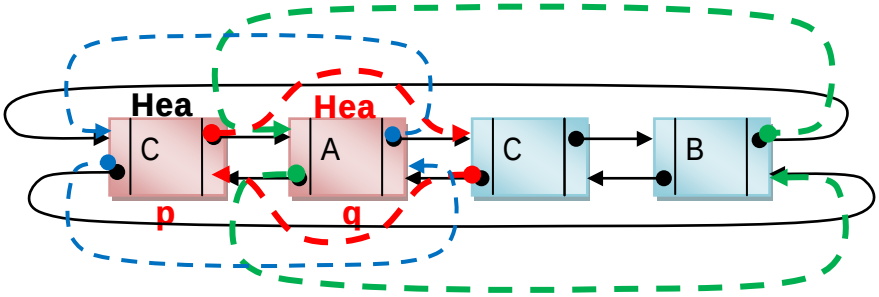
5.22 Ordenar burbuja intercambiando ligas

Diseñar un algoritmo que permita ordenar una lista doble circular con el método de la burbuja, intercambiando las ligas

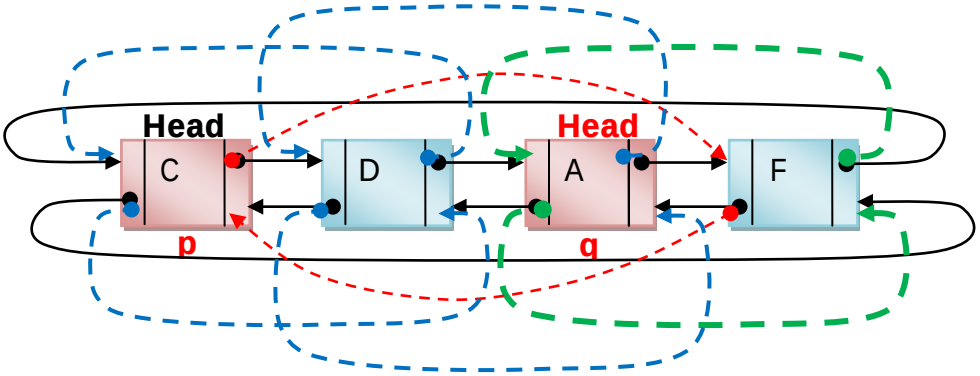
Análisis del problema

Para resolver este problema es necesario contar con al menos dos nodos. Al intercambiar las ligas los valores no cambiaran de localidad de memoria. El método de la burbuja funciona comparando el primer elemento de la lista con el resto de los elementos, en caso de que sea necesario se realiza el intercambio, posteriormente se avanza hacia el segundo elemento para compararlo con el resto y así sucesivamente hasta terminar de comparar todos los elementos. Es necesario considerar cuatro casos posibles cuando se realice un intercambio:

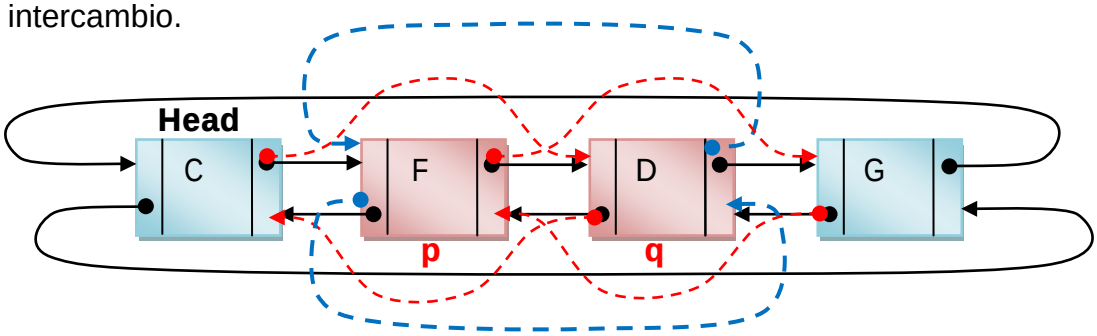
Caso 1: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.



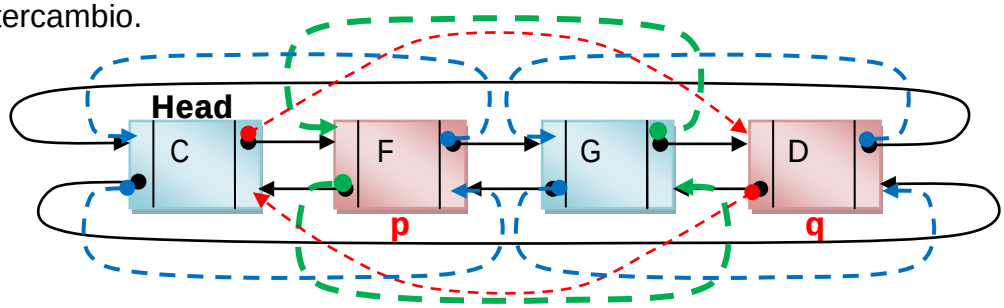
Caso 2: El nodo con el elemento mayor se encuentra al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Caso 3: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo está ligado con el que se va a realizar el intercambio.



Caso 4: El nodo con el elemento mayor se encuentra en una posición distinta al inicio de la lista y este nodo no está ligado con el que se va a realizar el intercambio.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras  p(sig) <>  null
        q ← p(sig)
        Mientras q <> null
            Si p(dato) > q(dato) entonces
                Si p = head y p(sig) = q entonces
                    p(sig) ← q(sig)
                    p(sig(ant)) ← p
                    q(sig) ← p
                    q(sig(ant)) ← q
                    head ← q
                    p ← q
                    q ← q(sig)
                Si p = head y p(sig) <> q entonces
                    r ← p(ant)
                    r(sig) ← p
                    r(sig(ant)) ← r
                    r ← p(sig)
                    p(sig) ← q(sig)
                    p(sig(ant)) ← p
                    q(sig) ← r
                    q(sig(ant)) ← q
                    head ← q
                    q ← p
                    p ← head
                Si p <> head y p(sig) = q entonces
                    r ← p(ant)
                    r(sig) ← q
                    r(sig(ant)) ← r
                    p(sig) ← q(sig)
                    p(sig(ant)) ← p
                    q(sig) ← p
                    q(sig(ant)) ← q
                    p ← q
                    q ← q(sig)
                Si p <> head y p(sig) <> q entonces
                    r ← p(ant)
                    s ← q(ant)
                    s(sig) ← p
                    s(sig(ant)) ← s
                    s ← p(sig)
                    p(sig) ← q(sig)
                    p(sig(ant)) ← p
                    q(sig) ← s
                    q(sig(ant)) ← q
                    r(sig) ← q
                    r(sig(ant)) ← r
                    p ← q
                    q ← q(sig)
            q ← q(sig)
        p ← p(sig)
De lo contrario
        Mensaje (Lista vacía)
```

Código en C++

```
void lista_doble_circular::burbuja_ligas()
{
    nodo p,q,r,s;
    if (head != NULL)
    {
      p = head;
      while (p->sig != head)
      {
          q = p->sig;
          while (q != head)
          {
              if (p->dato > q->dato)
              {
                  if (p==head && p->sig==q)
                  {
                      r=head->ant;
                      p->sig = q->sig;
                      p->sig->ant = p;
                      q->sig = p;
                      q->sig->ant = q;
                      head = q;
                      r->sig = head;
                      r->sig->ant = r;
                      p = q;
                      q = q->sig;
                  }
                  else
                  {
                    if (p==head && p->sig != q)
                    {
                      s=head->ant;
                      r = q->ant;
                      r->sig = p;
                      r->sig->ant = r;
                      r = p->sig;
                      if (s!=q)
                      {
                        p->sig = q->sig;
                        p->sig->ant = p;
                        q->sig = r;
```

```
                        q->sig->ant = q;
                        head = q;
                        s->sig = head;
                        s->sig->ant = s;
                        q = p;
                        p = head;
                    }
                    else
                    {
                        p->sig = q;
                        p->sig->ant = p;
                        q->sig = r;
                        q->sig->ant = q;
                        head = q;
                        q = p;
                        p = head;
                    }
                }
                else
                {
                    if (p!=head && p->sig == q)
                    {
                        r = p->ant;
                        r->sig = q;
                        r->sig->ant = r;
                        p->sig = q->sig;
                        p->sig->ant = p;
                        q->sig = p;
                        q->sig->ant = q;
                        p = q;
                        q = q->sig;
                    }
                    else
                    {
                        if (p!=head && p->sig != q)
                        {
                            r = p->ant;
                            s = q->ant;
                            s->sig = p;
                            s->sig->ant = s;
                            s = p->sig;
                            p->sig = q->sig;
                            p->sig->ant = p;
                            q->sig = s;
```

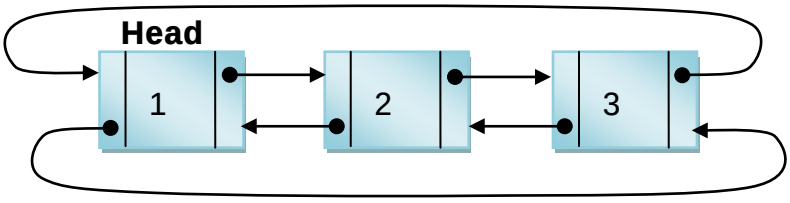
```
                                q->sig->ant = q;
                                r->sig = q;
                                r->sig->ant = r;
                                p = q;
                                q = q->sig;
                            }
                        }
                    }
                }
                q = q->sig;
            }
            p = p->sig;
        }
    }
    Else
        cout << "Lista vacía...";
}
```

5.23 Buscar posición

Diseñar un algoritmo que permita devolver en qué posición se encuentra un carácter en una lista doble circular.

Análisis del problema

Para resolver este problema es necesario contar con el valor del carácter que se buscará en la lista, recorrer toda la lista comparando el valor de cada nodo con el valor del carácter hasta encontrar el final de la lista o hasta que se encuentre el carácter.



Solución en pseudocódigo

```
Si head <> null entonces
    Leer (valor)
    p ← head
    pos ← 1
    Mientras (( p(sig) <>  head) o (p(dato)<>valor))
        p ← p(sig)
        pos ← pos + 1
    Si p(dato) = valor entonces
        Mensaje ('Posición = ', pos)
    De lo contrario
        Mensaje ('El valor no se encuentra en la lista')
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

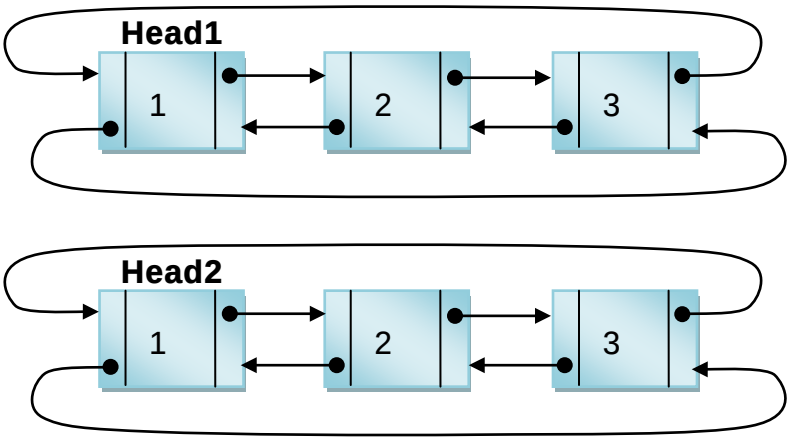
```
int lista_doble_circular::posicion(char valor)
{
   nodo p;
   int pos;
   if (head != NULL)
   {
       p = head;
       pos = 1;
       while (p->sig != head && p->dato != valor)
       {
           p = p->sig;
           pos++;
       }
       if (p->dato != valor)
          pos = 0;
    }
    return pos;
}
```

5.24 Comparar dos listas

Diseñar un algoritmo que permita comprobar si dos listas dobles circulares son exactamente iguales.

Análisis del problema

Para resolver este problema es necesario recorrer ambas listas al mismo tiempo para ir comparando nodo por nodo su valor.



Solución en pseudocódigo

```
Si head1 <> null y head2 <> null entonces
    p ← head1
    q ← head2
    igual ← verdadero
    Mientras igual = verdadero
        y p(sig)<> head y  q(sig) <>head
        Si p(dato) <> q(dato) entonces
            igual ← falso
        p ← p(sig)
        q ← q(sig)
    Si p(dato) <> q(dato) entonces
            igual ← falso
    Si igual = verdadero entonces
        Mensaje ('Listas iguales')
    De lo contrario
        Mensaje ('Las listas no son iguales')

De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

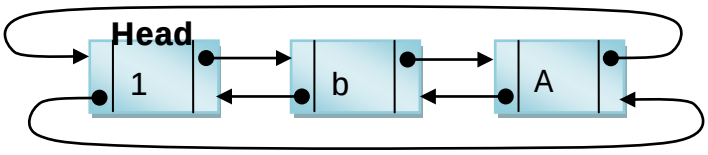
```
int lista_doble_circular::comparar(lista_doble_circular a)
{
   nodo p,q;
   int igual;
   if (head != NULL && a.head != NULL)
   {
     if (tamano() == a.tamano())
      {
       p = head;
       q = a.head;
       igual = true;
       do
       {
           if (p->dato != q->dato)
               igual = false;
           p = p->sig;
           q = q->sig;
       }while (igual && (p!= head && q!=head));
      }
     else
        igual = false;
   }
   else
       cout << "Listas vacías...";
   return igual;
}
```

5.25 Reemplazar texto

Diseñar un algoritmo que permita reemplazar parte de una lista doble circular por otro texto.

Análisis del problema

Para resolver este problema es necesario calcular el tamaño de la lista y el tamaño del texto que se va a reemplazar, para determinar si es posible que a partir de la posición que se indique se pueda llevar a cabo la operación. En caso de que los valores sean correctos, se ubica un apuntador en la posición en donde se hará el reemplazo y se inicia con el reemplazo.



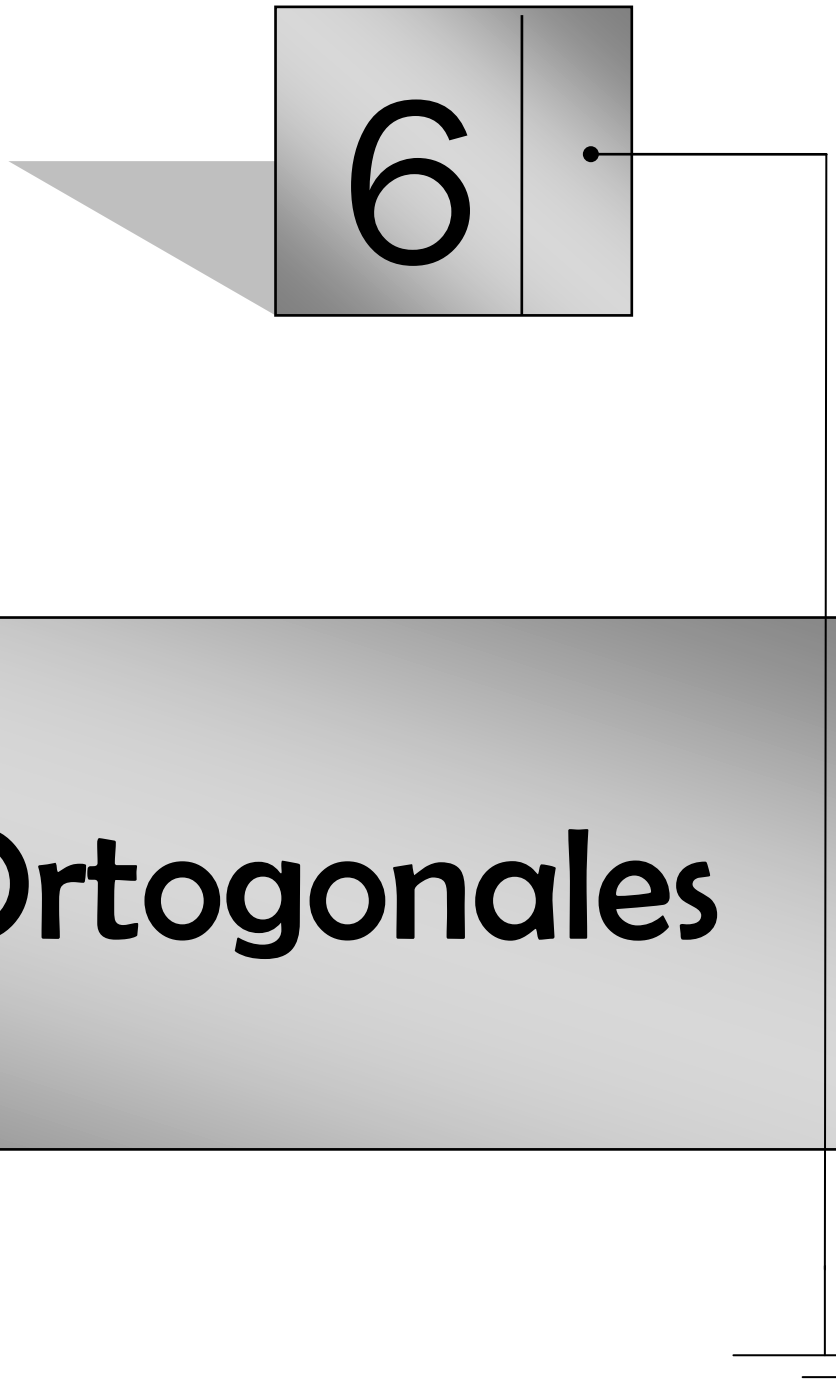
Solución en pseudocódigo

```
Si head <> null
    Leer (pos)
    Leer (texto)
    n ← tamaño(texto)
    c ← 1
    p ← head
    Mientras p(sig) <> head
            p ← p(sig)
            c ← c + 1
    Si (pos+n-1) <= c entonces
        p← head
        i = 1
        Mientras i <> pos entonces
            i ← i + 1
            p ← p(sig)
        j ← 1
        Mientras j <> n
            p(dato) ← texto[j]
            p ← p(sig)
            j ← j + 1
    De lo contrario
        Mensaje ('Error en los valores')
De lo contrario
    Mensaje (Lista vacía)
```

Código en C++

```
void lista_doble_circular::reemplazar(int pos, char *valor)
{
  nodo p;
  int n,c,i,j;
  if (head != NULL)
  {
    n = strlen(valor);
    c = 1;
    p = head;
    while (p->sig != head)
    {
        p = p->sig;
        c++;
    }
    if ((pos+n-1) <= c)
    {
        p = head;
        i = 1;
        while (i != pos)
        {
            i++;
            p = p->sig;
        }
        j = 0;
        while (j< n)
        {
            p->dato = valor[j];
            p = p->sig;
            j++;
        }
    }
    else
        cout << "Error en los valores...";
  }
  else
    cout << "Lista vacía...";
}
```

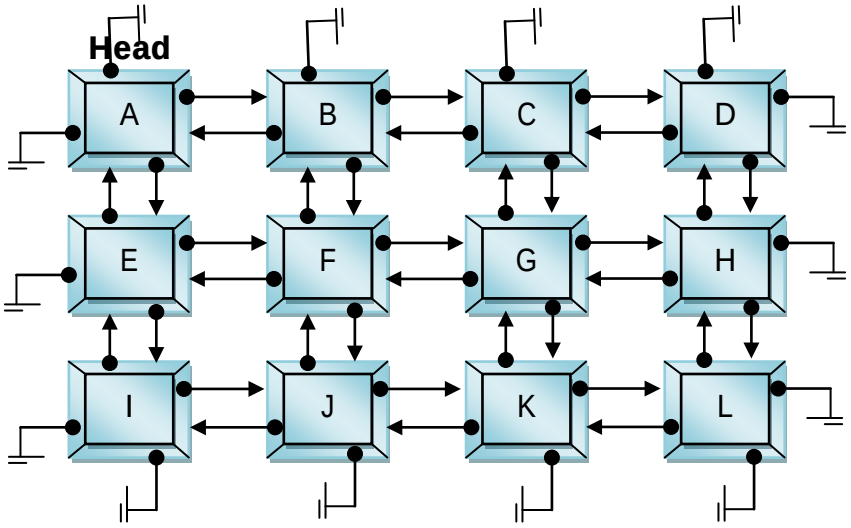
[Capítulo]



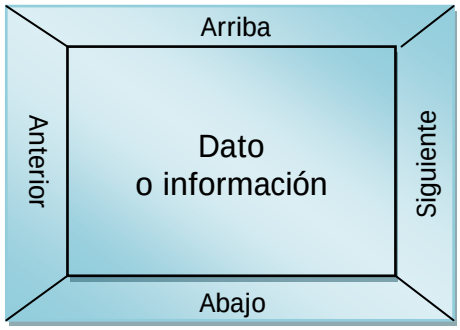
Listas Ortogonales

6 LISTAS ORTOGONALES

La característica principal de una lista ortogonal lineal es que las ligas de los últimos nodos apuntan hacia el valor nulo.



El nodo de una lista ortogonal debe contener como mínimo cinco campos: uno para almacenar la información y cuatro para guardar la dirección de memoria hacia el siguiente, anterior, arriba y abaja nodo de la lista. En la figura se puede apreciar la estructura del nodo para una lista ortogonal.



Para definir la estructura del nodo en C++ se hace lo siguiente:

```
struct apuntador
{
  char dato;
  apuntador *sig;
  apuntador *ant;
  apuntador *ar;
  apuntador *ab;
};
```

Para simplificar la asignación de memoria se utiliza la siguiente función:

```
nodo nuevo()
{
 nodo p;
 p = new struct apuntador;
 return p;
}
```

Se presenta la clase lista_ortogonal, la cual incluye la variable head y los métodos de las operaciones que se desarrollan en este capítulo para el manejo de las listas ortogonales.

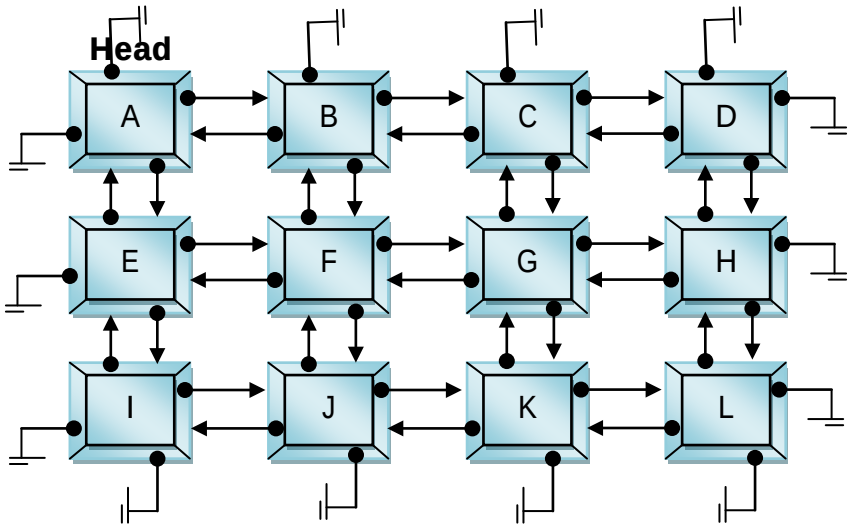
```
class lista_ortogonal
{
  nodo head;
  public:
   lista_ortogonal();
   void crear(int n, int m);
   void desplegar();
   int tamano();
   int renglores();
   int columnas();
   void insertar_renglon_final();
   void insertar_columna_final();
   void insertar_renglon(int pos);
   void insertar_columna(int pos);
   void insertar_renglon_inicio();
   void insertar_renglon_final();
   void borrar_renglon_final();
   void borrar_columna_final();
   void borrar_renglon_inicio();
   void borrar_columna_inicio();
   void borrar_renglon(int pos);
   void borrar_columna(int pos);
};
```

6.1 Crear una lista ortogonal

Diseñar un algoritmo que permita crear una lista ortogonal lineal con n renglones y m columnas.

Análisis del problema

Para resolver este problema es necesario conocer el valor de n y de m, para fijar los límites de los ciclos que estarán generando cada uno de los nodos que formaran parte de la lista. Es necesario introducir la información de cada uno de los nodos dentro del ciclo.



Solución en pseudocódigo

```
Leer (n)
Leer (m)
Para i = 1 hasta n
    Para j = 1  hasta m
        Nuevo(p)
        Leer(p(dato))
        p(sig) ← null
        p(ab) ← null
        Si j = 1 entonces
            p(ant) ← null
            Si head = null entonces
                Head ← p
            q ← p
        De lo contrario
            p(ant) ← q
            q(sig) ← p
            q ← p
        Si i = 1 entonces
            p(ar) ← null
            q ← p
        de lo contrario
            p(ar) ← r
            r(ab) ← p
            r ← r(sig)
    r ← head
    Mientras r(ab) <> null
        r ← r(ab)
```

Código en C++

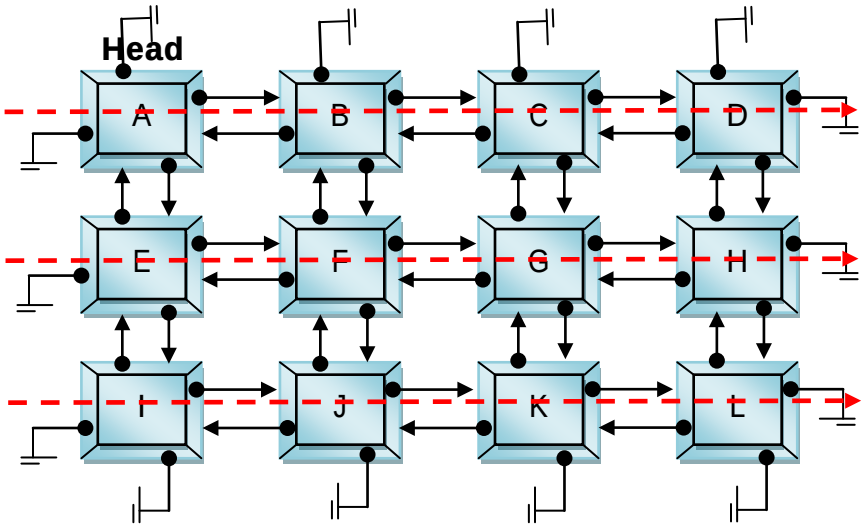
```
void lista_ortogonal::crear(int n, int m)
{
  nodo p,q,r;
  int i,j;
  for (i=1; i<= n; i++)
  {
      for (j=1; j <= m; j++)
      {
          p=nuevo();
          cout << "p(dato) = ";
          cin >> p->dato;
          p->sig = NULL;
          p->ab = NULL;
          if (j==1)
          {
              p->ant = NULL;
              if (head == NULL)
                head = p;
              q = p;
          }
          else
          {
              p->ant = q;
              q->sig = p;
              q = p;
          }
          if (i==1)
          {
              p->ar = NULL;
              q = p;
          }
          else
          {
              p->ar = r;
              r->ab = p;
              r = r->sig;
          }
      }
      r = head;
      while (r->ab != NULL)
        r = r->ab;
  }
 }
```

6.2 Recorrer la lista

Diseñar un algoritmo que permita desplegar el contenido de una lista ortogonal lineal por renglones.

Análisis del problema

Para resolver este problema es necesario recorrer la lista por renglones utilizando dos ciclos anidados. El ciclo mas externo recorre los renglones y el ciclo mas interno recorre las columnas.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p <> null
        q ← p
        Mientras q <> null
            desplegar(q(dato))
            q ← q(sig)
        p ← p(ab)
De lo contrario
    mensaje ('Lista vacía...')
```

Código en C++

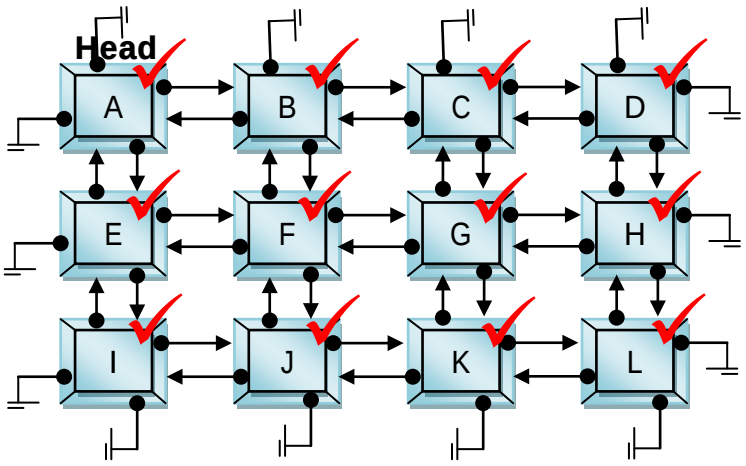
```
void lista_ortogonal::desplegar()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p  != NULL)
        {
            q = p;
            while (q != NULL)
            {
                cout << q->dato << "  ";
                q = q->sig;
            }
            cout << endl;
            p = p->ab;
        }
    }
    else
        cout << "Lista vacia...";
}
```

6.3 Tamaño de la lista

Diseñar un algoritmo que permita calcular el total de nodos de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario recorrer la lista ya sea por renglones o por columnas utilizando dos ciclos anidados, para contar cada uno de los nodos.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    n ← 0
    Mientras p <> null
        q ← p
        Mientras q <> null
            n ← n + 1
            q ← q(sig)
        p ← p(ab)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

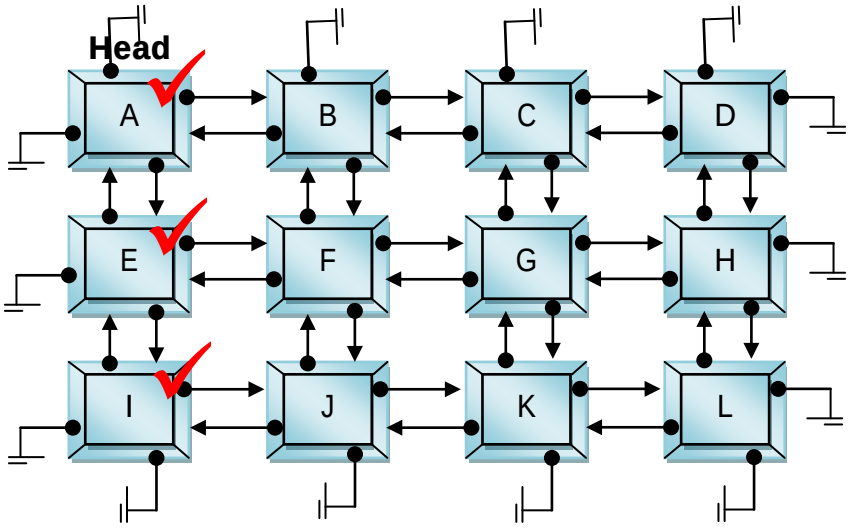
```
int lista_ortogonal::tamano()
{
    nodo p, q;
    int n;
    n = 0;
    if ( head != NULL)
    {
        p = head;
        while ( p != NULL)
        {
            q = p;
            while (q != NULL)
            {
                n++;
                q = q->sig;
            }
            p = p->ab;
        }
    }
    else
    {
        cout << "Lista vacía...";
    }
    return n;
}
```

6.4 Total de renglones

Diseñar un algoritmo que permita calcular el total de renglones de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario recorrer la primera de las columnas de la lista para ir contando cada uno de los renglones.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    n ← 0
    Mientras p <> null
        n ← n + 1
        p ← p(ab)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

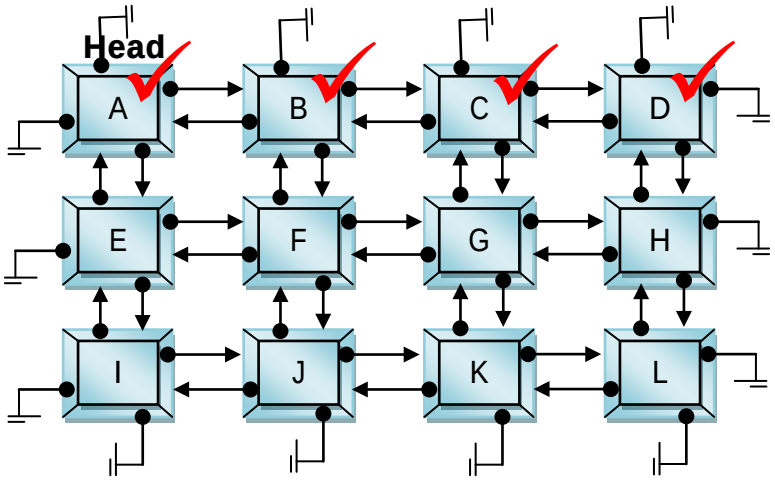
```
int lista_ortogonal::renglones()
{
    nodo p;
    int n;
    n = 0;
    if ( head != NULL)
    {
        p = head;
        while ( p != NULL)
        {
            n++;
            p = p->ab;
        }
    }
    return n;
}
```

6.5 Total de columnas

Diseñar un algoritmo que permita calcular el total de columnas de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario recorrer el primer renglón de la lista para ir contando cada una de las columnas.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    n ← 0
    Mientras p <> null
        n ← n + 1
        p ← p(sig)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

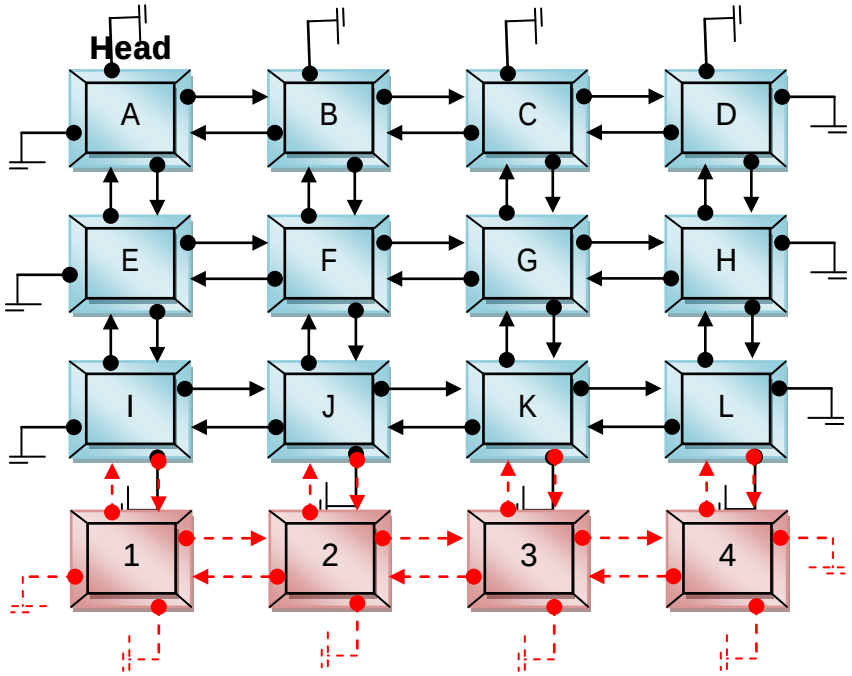
```
int lista_ortogonal::columnas()
{
    nodo p;
    int n;
    n = 0;
    if ( head != NULL)
    {
        p = head;
        while ( p != NULL)
        {
            n++;
            p = p->sig;
        }
    }
    return n;
}
```

6.6 Insertar renglón al final

Diseñar un algoritmo que permita insertar un renglón al final de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el último nodo de la primera columna que recorra el último renglón para ir ligando cada uno de los nodos del renglón que se inserta con la lista.



Solución en pseudocódigo

```
p ← head
Si pos <= renglones + 1 entonces
    Si pos = 1 entonces
        q ← p
        mientras q <> null
            Nuevo(r)
            Leer(r(dato))
            r(sig) ← null
            r(ar) ← null
            r(ab) ← q
            r(ab(ar)) ← r
            Si q(ant) = null entonces
                r(ant) ← null
                head ← r
            De lo contrario
                r(ant) ← q(ant(ar))
                r(ant(sig)) ← r
            q ← q(sig)
    De lo contrario
        n ← 1
        Mientras (n < pos -1 )
            n ← n + 1
            p ← p(ab)
        q ← p
        Mientras q <> null
            Nuevo(r)
            Leer (r(dato))
            r(sig) ← null
            r(ar) ← q
            r(ab) ← q(ab)
            r(ar(ab)) ← r
            Si r(ab) <> null entonces
                r(ab(ar)) ← r
            Si q(ant) = null entonces
                r(ant) ← null
            De lo contrario
                r(ant) ← q(ant(ab))
                r(ant(sig)) ← r
            q ← q(sig)
De lo contrario
    Mensaje ('Posición inválida...')
```

Código en C++

```
void lista_ortogonal::insertar_renglon(int pos)
{
    nodo p,q,r;
    int n;
    p = head;
    if (pos <= renglones()+1)
    {

        if (pos == 1)
        {
            q = p;
            while (q!= NULL)
            {
                r=nuevo();
                cout << "r(dato) = ";
                cin >> r->dato;
                r->sig = NULL;
                r->ar = NULL;
                r->ab = q;
                r->ab->ar = r;
                if (q->ant == NULL)
                {
                    r->ant = NULL;
                    head = r;
                }
                else
                {
                    r->ant = q->ant->ar;
                    r->ant->sig = r;
                }
                q = q->sig;
            }
        }

       else
        {
            n = 1;
            while (n < pos-1)
            {
               n++;
               p = p->ab;
            }
```

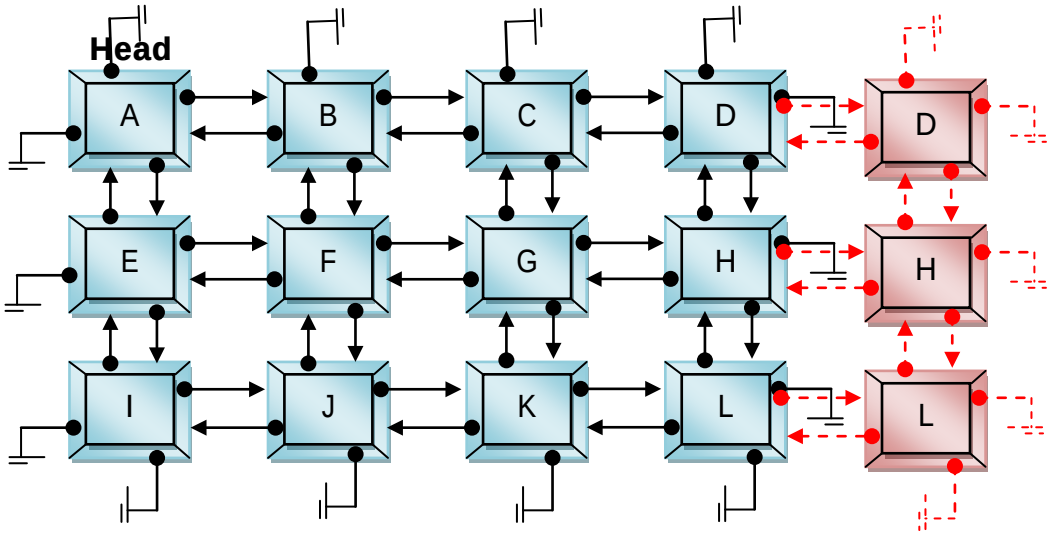
```
                q = p;
                while (q!= NULL)
                {
                    r=nuevo();
                    cout << "r(dato) = ";
                    cin >> r->dato;
                    r->sig = NULL;
                    r->ar = q;
                    r->ab = q->ab;
                    r->ar->ab = r;
                    if (r->ab != NULL)
                        r->ab->ar = r;
                    if (q->ant == NULL)
                        r->ant = NULL;
                    else
                    {
                        r->ant = q->ant->ab;
                        r->ant->sig = r;
                    }
                    q = q->sig;
                }
            }
        }
        else
            cout << "Posicion invalida... ";
}
```

6.7 Insertar columna al final

Diseñar un algoritmo que permita insertar una columna al final de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el primer nodo de la última columna de la lista para ir ligando cada uno de los nuevos nodos de la columna.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(sig) <> null
        p ← p(sig)
    Mientras p <> null
        Nuevo(q)
        Leer (q(dato))
        q(sig) ← null
        q(ant) ← p
        p(sig) ← q
        q(ab) ← null
        Si p(ar) = null entonces
            q(ar) ← null
        De lo contrario
            q(ar) ← p(ar(sig))
            q(ar(ab)) ← q
De lo contrario
    Nuevo (p)
    Leer(p(dato))
    head ← p
    p(ant) ← null
    p(ab) ← null
    p(sig) ← null
    p(ar) ← null
```

Código en C++

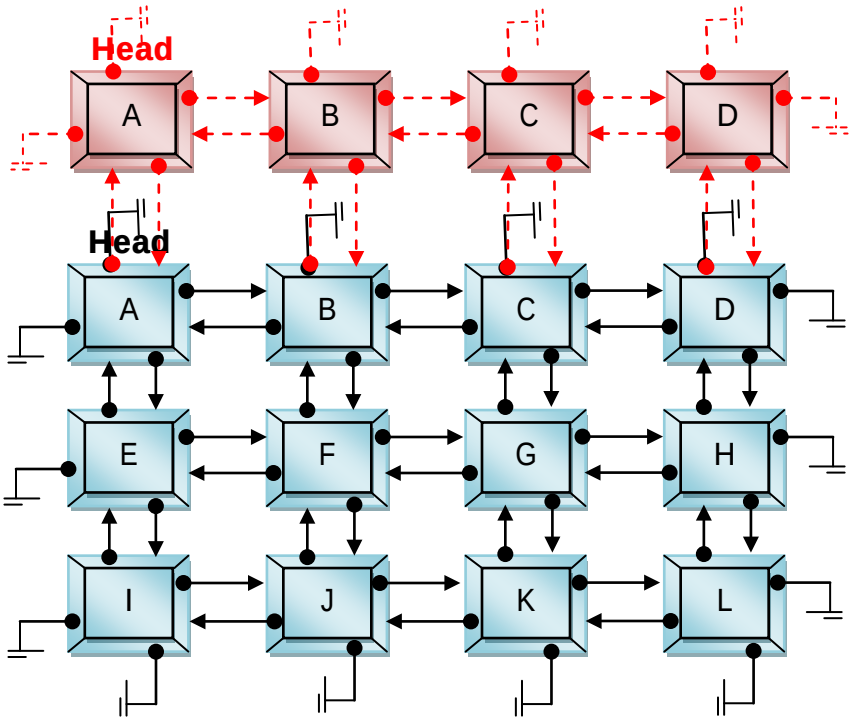
```
void lista_ortogonal::insertar_columna_final()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p->sig != NULL)
            p = p->sig;
        while (p!= NULL)
        {
            q=nuevo();
            cout << "q(dato) = ";
            cin >> q->dato;
            q->sig = NULL;
            q->ant = p;
            p->sig = q;
            q->ab = NULL;
            if (p->ar == NULL)
            {
                q->ar = NULL;
            }
            else
            {
                q->ar = p->ar->sig;
                q->ar->ab = q;
            }
            p = p->ab;
        }
    }
    else
    {
          p=nuevo();
          cout << "p(dato) = ";
          cin >> p->dato;
          head = p;
          p->ant = NULL;
          p->ab = NULL;
          p->sig = NULL;
          p->ar = NULL;
    }
}
```

6.8 Insertar renglón al inicio

Diseñar un algoritmo que permita insertar un renglón al inicio de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el primer nodo de la lista para recorrer el renglón. Durante el recorrido se va creando y ligando cada uno de los nuevos nodos del renglón.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p <> null
        Nuevo(q)
        Leer(q(dato))
        q(sig) ← null
        q(ab) ← p
        q(ar) ← null
        p(ar) ← q
    Si  p(ant) = null entonces
        q(ant) ← null
        head ← q
    De lo contrario
        q(ant) ← p(ant(ar))
        q(ant(sig)) q
    p ← p(sig)
De lo contrario
    Nuevo(p)
    Leer(p(dato))
    head ← p
    p(ant) ← null
    p(ab) ← null
    p(sig) ← null
    p(ar) ← null
```

Código en C++

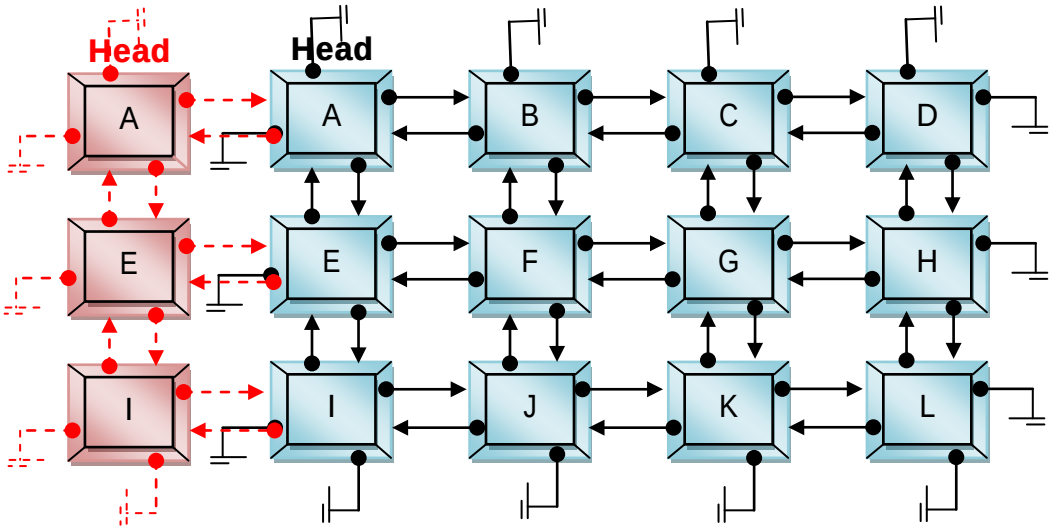
```
void lista_ortogonal::insertar_renglon_inicio()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p!= NULL)
        {
            q=nuevo();
            cout << "q(dato) = ";
            cin >> q->dato;
            q->sig = NULL;
            q->ab = p;
            q->ar = NULL;
            p->ar = q;
            if (p->ant == NULL)
            {
                q->ant = NULL;
                head = q;
            }
            else
            {
                q->ant = p->ant->ar;
                q->ant->sig = q;
            }
            p = p->sig;
        }
    }
   else
    {
          p=nuevo();
          cout << "p(dato) = ";
          cin >> p->dato;
          head = p;
          p->ant = NULL;
          p->ab = NULL;
          p->sig = NULL;
          p->ar = NULL;
    }
}
```

6.9 Insertar columna al inicio

Diseñar un algoritmo que permita insertar una columna al inicio de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el primer nodo de la lista para recorrer la primera columna. Durante el recorrido se va creando y ligando cada uno de los nuevos nodos de la columna.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p<> null
        Nuevo(q)
        Leer (q(dato))
        q(sig) ← p
        q(ab) ← null
        q(ant) ← null
        p(ant) ← q
        Si p(ar) = null entonces
            q(ar) ← null
            head ← q
        De lo contrario
            q(ar) ← p(ar(ant))
            q(ar(ab)) ← q
        p ← p(ab)
De lo contrario
    Nuevo(p)
    Leer (p(dato))
    head ← p
    p(ant) ← null
    p(ab) ← null
    p(sig) ← null
    p(ar) ← null
```

Código en C++

```
void lista_ortogonal::insertar_columna_inicio()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p!= NULL)
        {
            q=nuevo();
            cout << "q(dato) = ";
            cin >> q->dato;
            q->sig = p;
            q->ab = NULL;
            q->ant = NULL;
            p->ant = q;
            if (p->ar == NULL)
            {
                q->ar = NULL;
                head = q;
            }
            else
            {
                q->ar = p->ar->ant;
                q->ar->ab = q;
            }
            p = p->ab;
        }
    }
    else
    {
          p=nuevo();
          cout << "p(dato) = ";
          cin >> p->dato;
          head = p;
          p->ant = NULL;
          p->ab = NULL;
          p->sig = NULL;
          p->ar = NULL;
    }
}
```

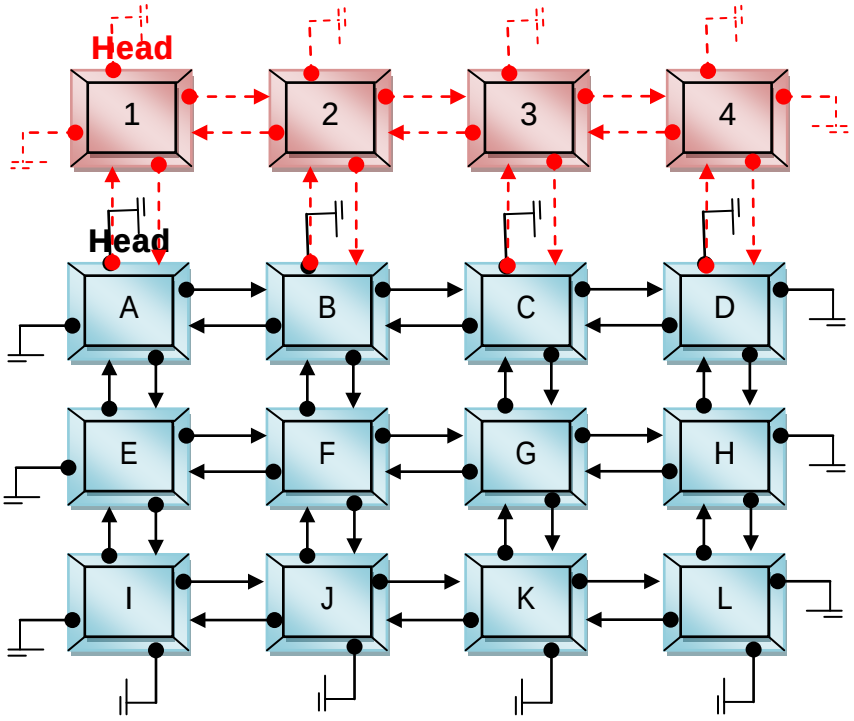
6.10 Insertar renglón en cualquier posición

Diseñar un algoritmo que permita insertar un renglón en cualquier posición en una lista ortogonal lineal.

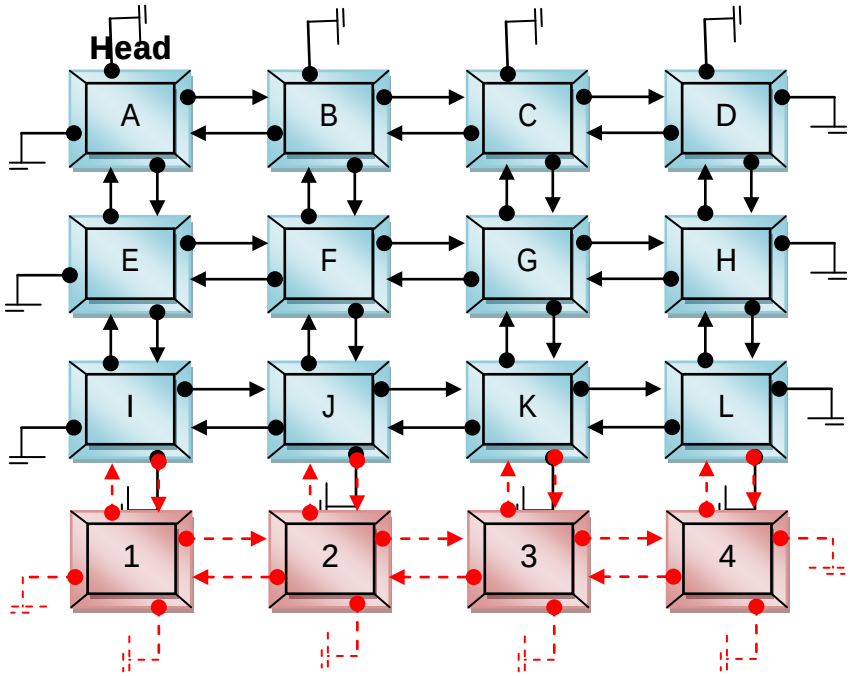
Análisis del problema

Para resolver este problema es necesario conocer la posición donde se ubicará el nuevo renglón. Es necesario validar la posición. Se ubica un apuntador en el primer nodo de la lista y se avanza en la primera columna hasta encontrar la ubicación donde se insertará el nuevo renglón. Para la solución de este problema se tienen que considerar los casos siguientes:

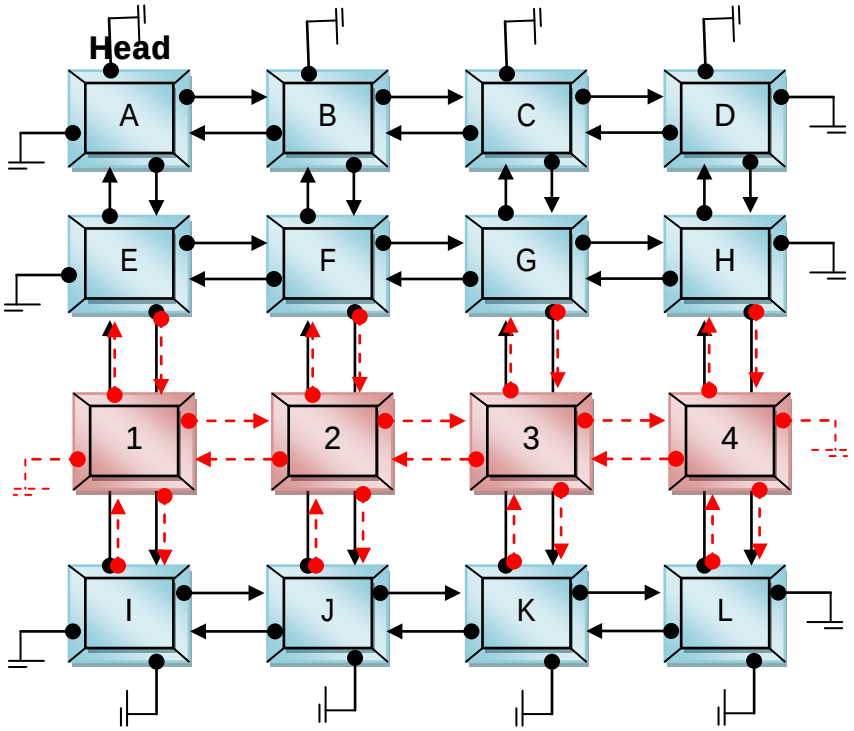
Caso 1: El renglón se inserta al inicio de la lista.



Caso 2: El renglón se inserta en la última posición de la lista.



Caso 3: El renglón se inserta en una posición intermedia dentro de la lista.



Solución en pseudocódigo

```
p ← head
Si pos <= renglones + 1 entonces
    Si pos = 1 entonces
        q ← p
        Mientras q <> null
            Nuevo(r)
            Leer (r(dato))
            r(sig) ← null
            r(ar) ← null
            r(ab) ← q
            r(ab(ar)) ← r
            Si q(ant) = null entonces
                r(ant) ← null
                head ← r
            De lo contrario
                r(ant) ← q(ant(ar))
                r(ant(sig)) ← r
            q ← q(sig)
    De lo contrario
        n ← 1
        Mientras n < pos-1
            n ← n+1
            p ← p(ab)
        q ← p
        Mientras q <> null
            Nuevo(r)
            Leer (r(dato))
            r(sig) ← null
            r(ar) ← q
            r(ab) ← q(ab)
            r(ar(ab)) ← r
            Si r(ab) <> null entonces
                r(ab(ar)) ← r
            Si q(ant) = null entonces
                r(ant) ← null
            De lo contrario
                r(ant) ← q(ant(ab))
                r(ant(sig)) ← r
            q ← q(sig)
De lo contrario
    Mensaje ('Posición inválida...')
```

Código en C++

```
void lista_ortogonal::insertar_renglon(int pos)
{
    nodo p,q,r;
    int n;
    p = head;
    if (pos <= renglones()+1)
    {
        if (pos == 1)
        {
            q = p;
            while (q!= NULL)
            {
                r=nuevo();
                cout << "r(dato) = ";
                cin >> r->dato;
                r->sig = NULL;
                r->ar = NULL;
                r->ab = q;
                r->ab->ar = r;
                if (q->ant == NULL)
                {
                    r->ant = NULL;
                    head = r;
                }
                else
                {
                    r->ant = q->ant->ar;
                    r->ant->sig = r;
                }
                q = q->sig;
            }
        }
        else
        {
            n = 1;
            while (n < pos-1)
            {
               n++;
               p = p->ab;
            }
            q = p;
            while (q!= NULL)
```

```
            {
                r=nuevo();
                cout << "r(dato) = ";
                cin >> r->dato;
                r->sig = NULL;
                r->ar = q;
                r->ab = q->ab;
                r->ar->ab = r;
                if (r->ab != NULL)
                    r->ab->ar = r;
                if (q->ant == NULL)
                    r->ant = NULL;
                else
                {
                    r->ant = q->ant->ab;
                    r->ant->sig = r;
                }
                q = q->sig;
            }
        }
    }
    else
        cout << "Posicion invalida... ";
}
```

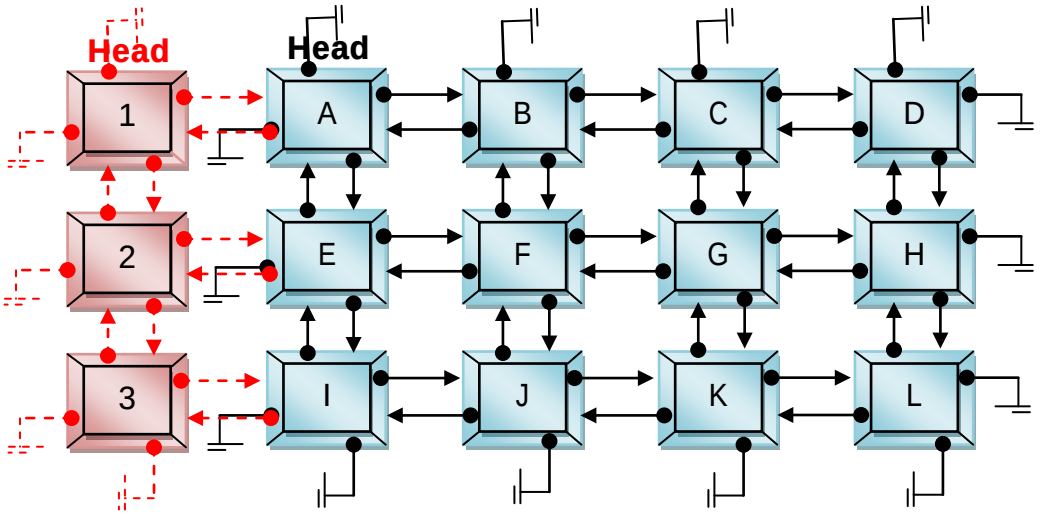
6.11 Insertar columna en cualquier posición

Diseñar un algoritmo que permita insertar una columna en cualquier posición en una lista ortogonal lineal.

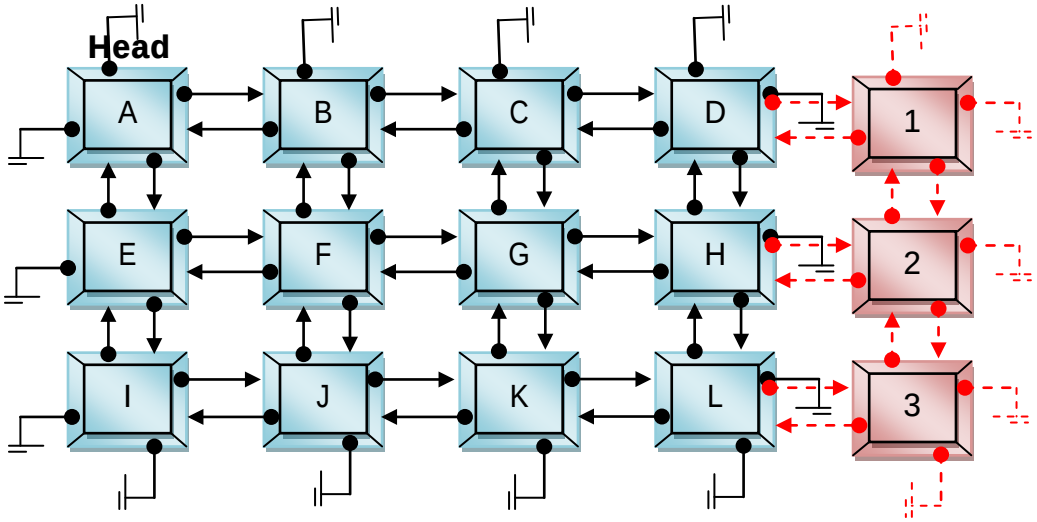
Análisis del problema

Para resolver este problema es necesario conocer la posición donde se ubicará la nueva columna. Es necesario validar la posición. Se ubica un apuntador en el primer nodo de la lista y se avanza en el primer renglón hasta encontrar la ubicación donde se insertará la nueva columna. Para la solución de este problema se tienen que considerar los casos siguientes:

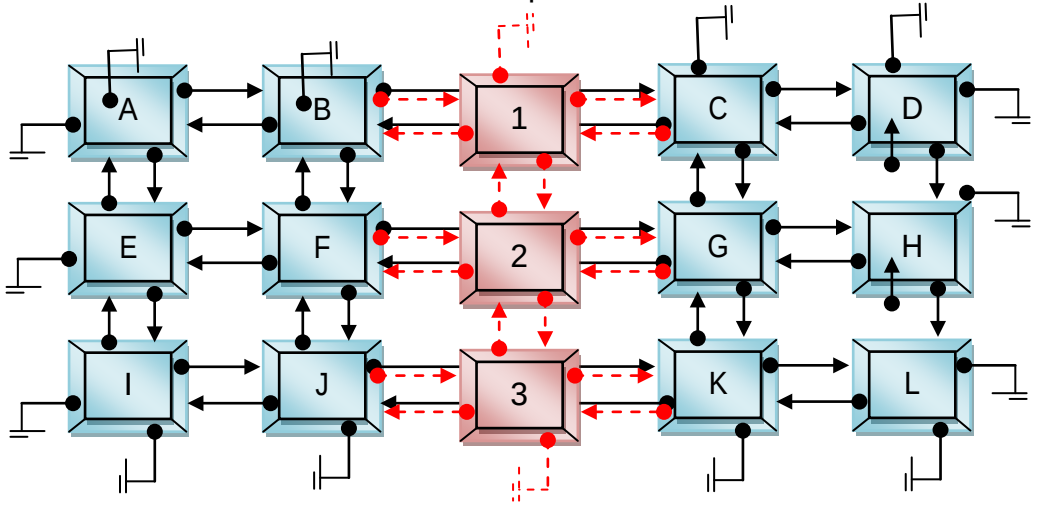
Caso 1: La columna se inserta al inicio de la lista.



Caso 2: La columna se inserta en la última posición de la lista.

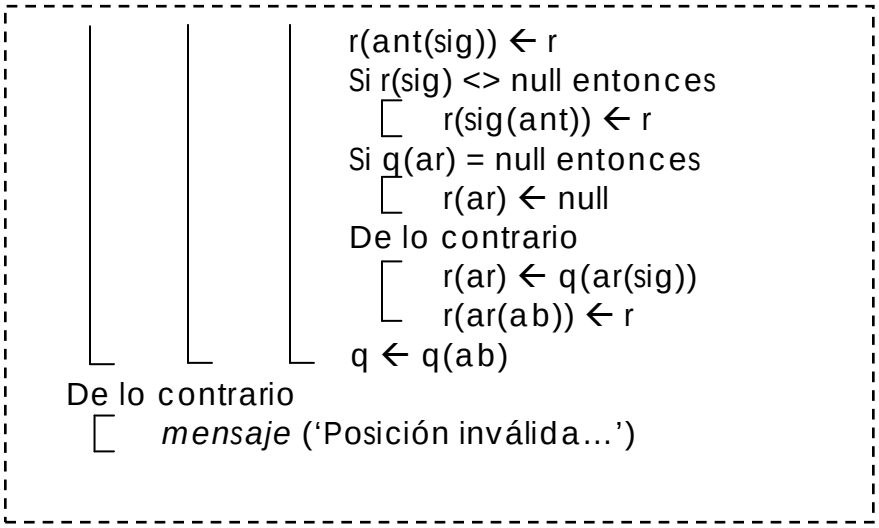


Caso 3: La columna se inserta en una posición intermedia dentro de la lista.



Solución en pseudocódigo

```
p ← head
Si pos <= columnas + 1 entonces
    Si pos = 1 entonces
        Si head <> null entonces
            q ← p
            Mientras q <> null
                Nuevo(r)
                Leer (r(dato))
                r(ant) ← null
                r(ab) ← null
                r(sig) ← q
                r(sig(ant)) ← r
                Si q(ar) = null entonces
                    r(ar) ← null
                    head ← r
                De lo contario
                    r(ar) ← q(ar(ant))
                    r(ar(ab)) ← r
                q ← q(ab)
        De lo contrario
            Nuevo(r)
            Leer (r(dato))
            r(ant) ← null
            r(ab) ← null
            r(ar) ← null
            r(sig) ← null
            head ← r
    De lo contrario
        n ← 1
        Mientras n < pos -1
            n ← n +1
            p ← p(sig)
        q ← p
        Mientras q <> null
            Nuevo(r)
            Leer (r(dato))
            r(ab) ← null
            r(ant) ← q
            r(sig) ← q(sig)
```



Código en C++

```
void lista_ortogonal::insertar_columna(int pos)
{
    nodo p,q,r;
    int n;
    p = head;
    if (pos <= columnas()+1)
    {

        if (pos == 1)
        {
            if (head != NULL)
            {
                q = p;
                while (q!= NULL)
                {
                    r=nuevo();
                    cout << "r(dato) = ";
                    cin >> r->dato;
                    r->ant = NULL;
                    r->ab = NULL;
                    r->sig = q;
                    r->sig->ant = r;
                    if (q->ar == NULL)
                    {
                        r->ar = NULL;
                        head = r;
```

```
                    }
                    else
                    {
                        r->ar = q->ar->ant;
                        r->ar->ab = r;
                    }
                    q = q->ab;
                }
            }
            else
            {
                r=nuevo();
                cout << "r(dato) = ";
                cin >> r->dato;
                r->ant = NULL;
                r->ab = NULL;
                r->ar = NULL;
                r->sig = NULL;
                head = r;
            }
        }
        else
        {
            n = 1;
            while (n < pos-1)
            {
               n++;
               p = p->sig;
            }
            q = p;
            while (q!= NULL)
            {
                r=nuevo();
                cout << "r(dato) = ";
                cin >> r->dato;
                r->ab = NULL;
                r->ant = q;
                r->sig = q->sig;
                r->ant->sig = r;
                if (r->sig != NULL)
                    r->sig->ant = r;
                if (q->ar == NULL)
                    r->ar = NULL;
```

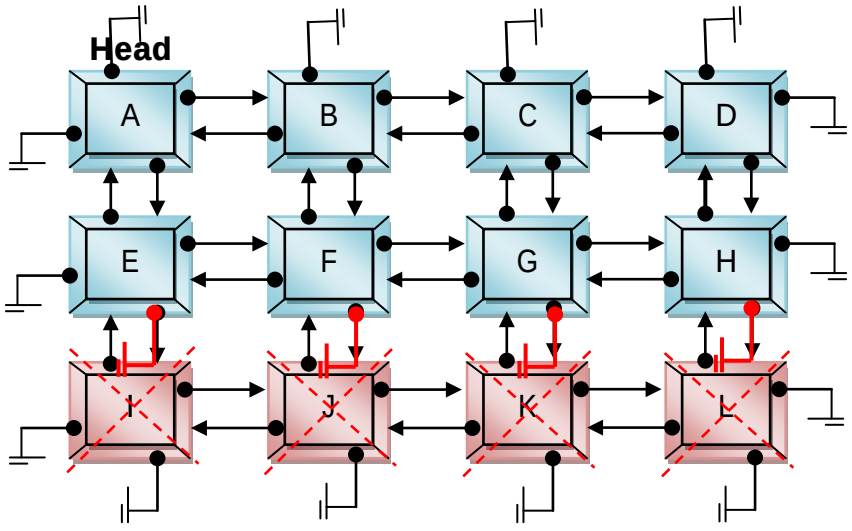
```
                    else
                    {
                        r->ar = q->ar->sig;
                        r->ar->ab = r;
                    }
                    q = q->ab;
                }
            }
        }
        else
            cout << "Posicion invalida... ";
}
```

6.12 Eliminar último renglón

Diseñar un algoritmo que permita eliminar el último renglón de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el primer nodo de la lista para recorrer la primera columna y posicionarse en el renglón que se va a eliminar.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(ab) <> null
        p ← p(ab)
    Si p <> head entonces
        q ← p(ar)
        Mientras q <> null
            Eliminar (q(ab))
            q(ab) ← null
            q ← q(sig)
    De lo contrario
        Mientras p <> null
            head ← head(sig)
            Eliminar (p)
            p ← head
De lo contrario
    mensaje ('Lista vacía...´)
```

Código en C++

```
void lista_ortogonal::borrar_renglon_final()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p->ab != NULL)
            p = p->ab;
        if (p != head)
        {
            q = p->ar;
            while (q!= NULL)
            {
                delete q->ab;
                q->ab = NULL;
                q = q->sig;
            }
```

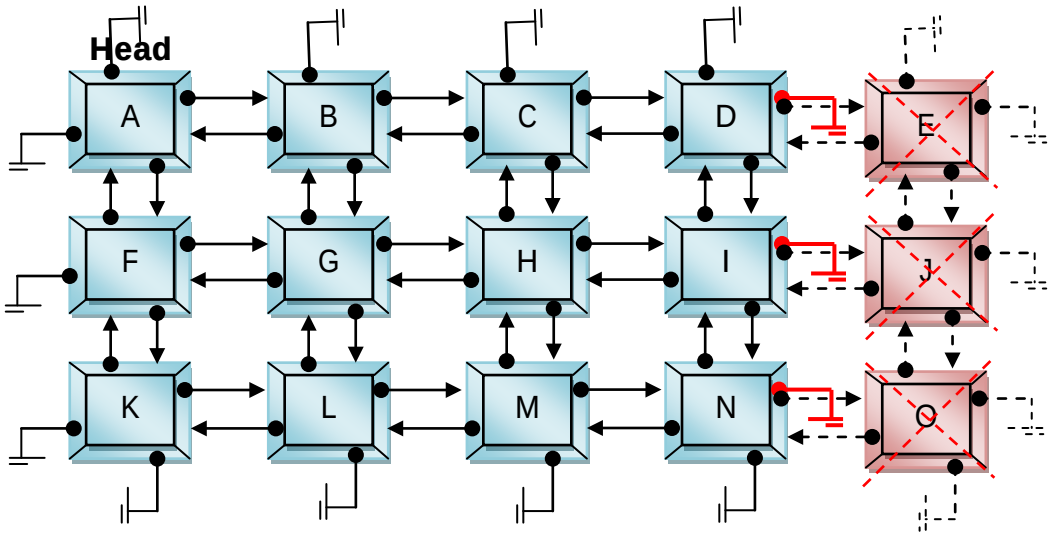
```
        }
        else
        {
            while (p!= NULL)
            {
                head = head->sig;
                delete(p);
                p = head;
            }
        }
    }
    else
        cout << "Lista vacía...";
}
```

6.13 Eliminar última columna

Diseñar un algoritmo que permita eliminar la última columna de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el primer nodo de la lista para recorrer la primera columna. Durante el recorrido se va creando y ligando cada uno de los nuevos nodos de la columna.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Mientras p(sig) <> null
        p ← p(sig)
    Si p <> head entonces
        q ← p(ant)
        Mientras q <> null
            Eliminar (q(sig))
            q(sig) ← null
            q ← q(ab)
    De lo contrario
        Mientras p <> null
            head ← head(ab)
            Eliminar (p)
            p ← head
De lo contrario
    Mensaje ('Lista vacía…´)
```

Código en C++

```
void lista_ortogonal::borrar_columna_final()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        while (p->sig != NULL)
            p = p->sig;
        if (p!= head)
        {
            q = p->ant;
            while (q!= NULL)
            {
                delete q->sig;
                q->sig = NULL;
                q = q->ab;
            }
        }
        else
```

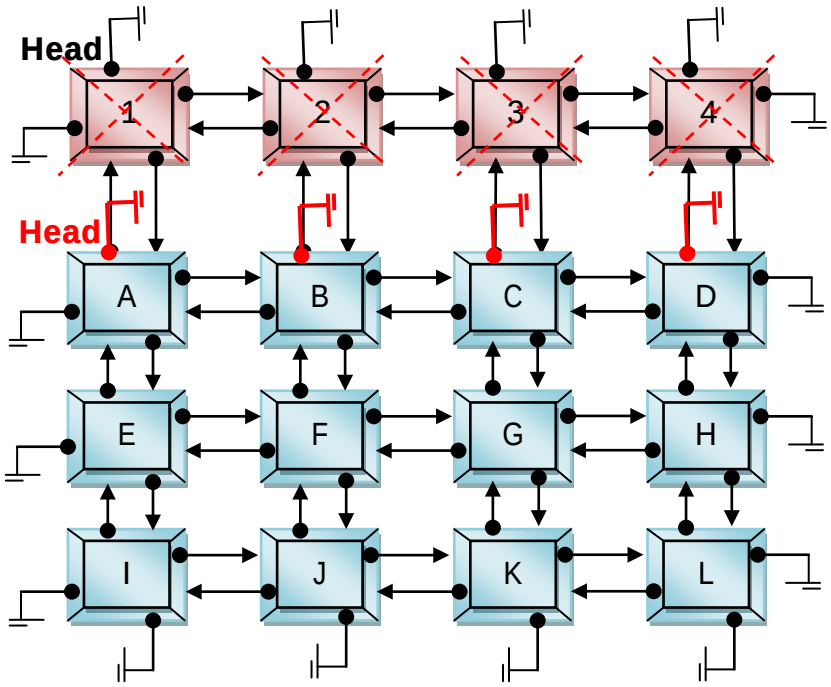
```
        {
            while (p!= NULL)
            {
                head = head->ab;
                delete(p);
                p = head;
            }
        }
    }
    else
        cout << "Lista vacía...";
}
```

6.14 Eliminar primer renglón

Diseñar un algoritmo que permita eliminar el primer renglón de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el primer nodo de la lista para recorrer el primer renglón. Durante el recorrido se va eliminando cada uno de los nodos del renglón. Si la lista tiene más de un renglón el head se mueve al primer nodo del segundo renglón, en caso contrario el head se inicializa con el valor nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Si renglones > 0 entonces
        q ← head(ab)
        Si q <> null entonces
            mientras q <> null
                q(ar) ← null
                Si p = head entonces
                    head ← q
                Eliminar (p)
                q ← q(sig)
                Si q <> null entonces
                    p ← q(ar)
        De lo contrario
            mientras p <> null
                head ← head(sig)
                Eliminar (p)
                p ← head
De lo contrario
    mensaje ('Lista vacía...')
```

Código en C++

```
void lista_ortogonal::borrar_renglon_inicio()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        if ( renglones() > 0 )
        {
            q = head->ab;
            if (q != NULL)
            {
                while (q != NULL)
                {
                    q->ar = NULL;
```

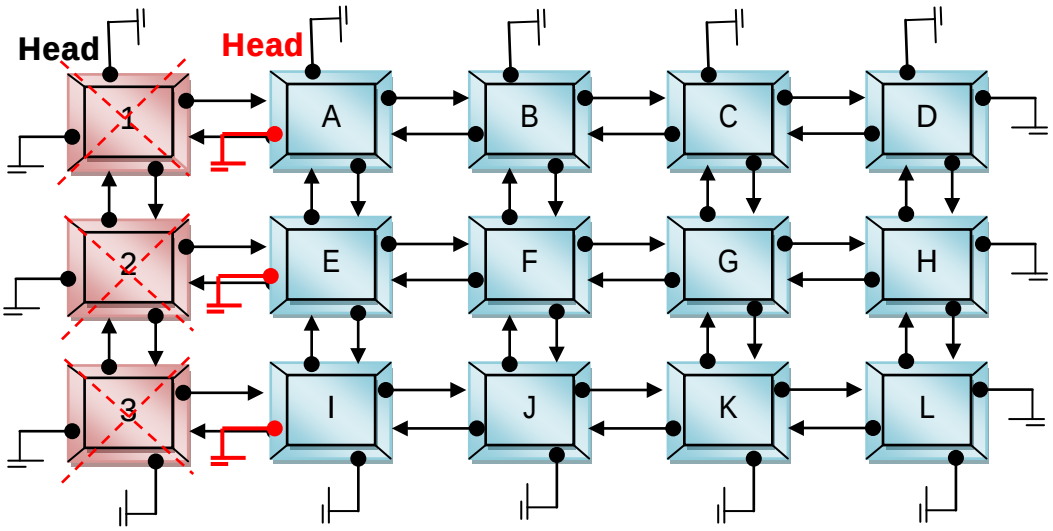
```
                        if (p==head)
                        {
                            head = q;
                        }
                        delete (p);
                        q = q->sig;
                        if ( q != NULL)
                            p = q->ar;
                    }
                }
                else
                {
                    while (p!= NULL)
                    {
                        head = head->sig;
                        delete(p);
                        p = head;
                    }
                }
            }
        }
        else
            cout << "Lista vacía...";
}
```

6.15 Eliminar primera columna

Diseñar un algoritmo que permita eliminar la primera columna de una lista ortogonal lineal.

Análisis del problema

Para resolver este problema es necesario ubicar un apuntador en el primer nodo de la lista para recorrer la primera columna. Durante el recorrido se va eliminando cada uno de los nodos de la columna. Si la lista tiene al menos dos columnas el head se mueve al primer nodo de la segunda columna, en caso contrario el head se inicializa con el valor nulo.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Si columnas > 0 entonces
        q ← head(sig)
        Si q <> null entonces
            Mientras q <> null
                q(ant) ← null
                Si p = head entonces
                    head ← q
                eliminar (p)
                q ← q(ab)
                Si q <> null entonces
                    p ← q(ant)
        De lo contrario
            Mientras p <> null
                head ← head(ab)
                eliminar (p)
                p ← head
De lo contrario
    mensaje ('Lista vacía...')
```

Código en C++

```
void lista_ortogonal::borrar_columna_inicio()
{
    nodo p,q;
    if (head != NULL)
    {
        p = head;
        if (columnas() > 0)
        {
            q = head->sig;
            if (q != NULL)
            {
               while (q != NULL)
               {
                    q->ant = NULL;
                    if (p==head)
                    {
                        head = q;
                    }
                    delete (p);
                    q = q->ab;
                    if ( q != NULL)
                        p = q->ant;
                }
            }
            else
            {
                while (p!= NULL)
                {
                    head = head->ab;
                    delete(p);
                    p = head;
                }
            }
        }
    }
    else
        cout << "Lista vacía...";
}
```

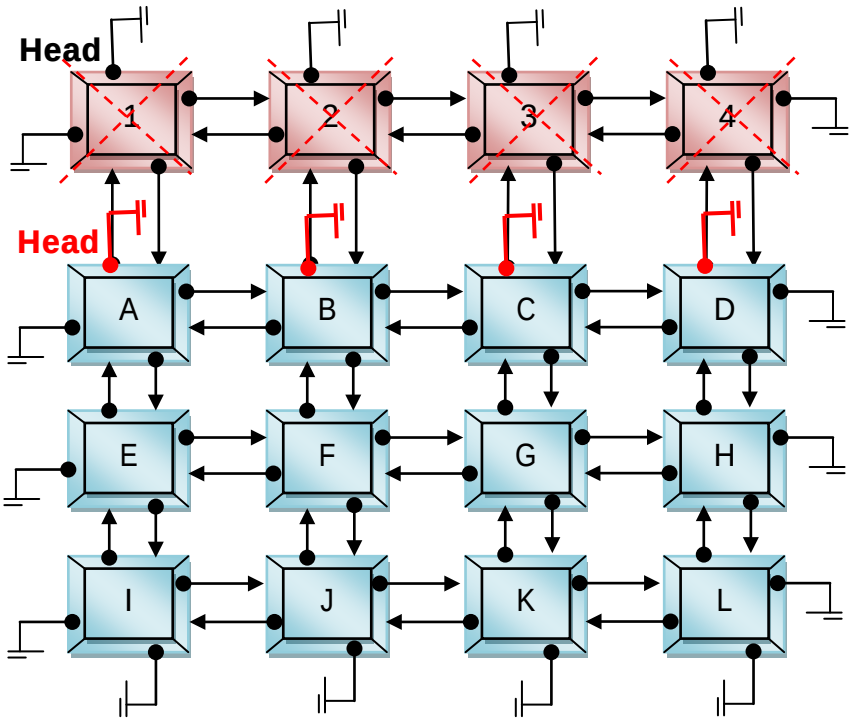
6.16 Eliminar cualquier renglón

Diseñar un algoritmo que permita eliminar cualquier renglón de una lista ortogonal lineal.

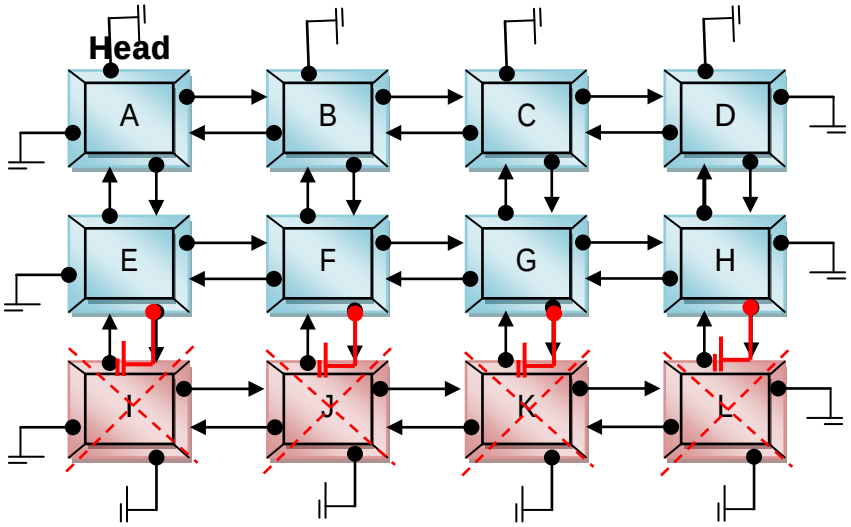
Análisis del problema

Para resolver este problema es necesario conocer y validar la posición del renglón que se eliminará. Se ubica un apuntador en el primer nodo de la lista y se avanza en la primera columna hasta encontrar la ubicación correcta del renglón, posteriormente se recorre todo el renglón para ir eliminando nodo por nodo. Para la solución de este problema se tienen que considerar los casos siguientes:

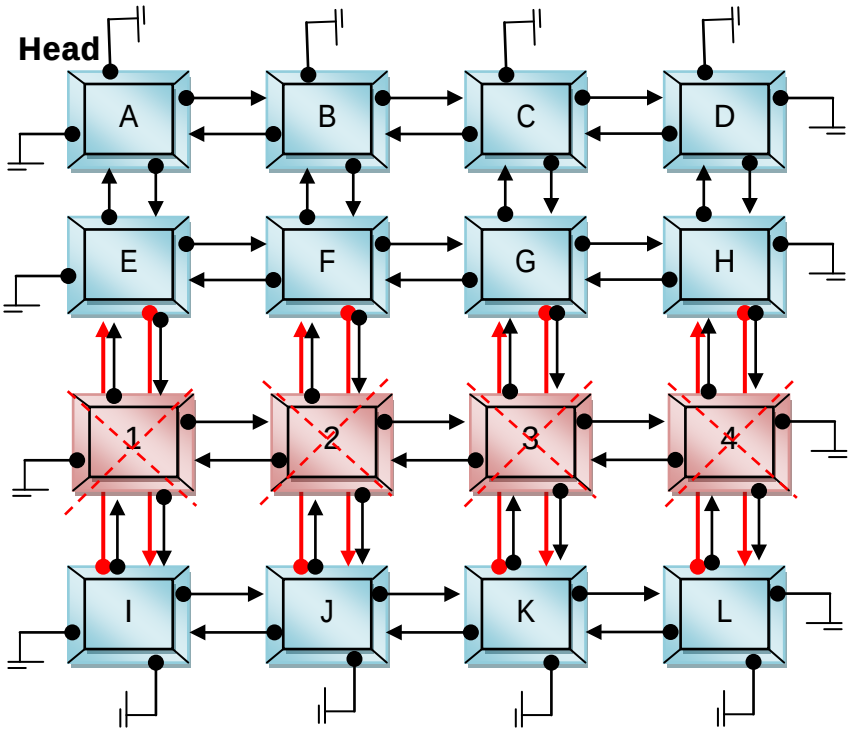
Caso 1: El renglón se encuentra ubicado al inicio de la lista.



Caso 2: El renglón se encuentra ubicado en la última posición de la lista.



Caso 3: El renglón se encuentra ubicado en una posición intermedia dentro de la lista.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Si pos <= renglones entonces
        Si pos = 1 entonces
            q ← head(ab)
            Si q <> null entonces
                Mientras q <> null
                    q(ar) ← null
                    Si p = head entonces
                        head ← q
                    Eliminar (p)
                    q ← q(sig)
                    Si q <> null entonces
                        p ← q(ar)
            De lo contrario
                Mientras p <> null
                    head ← head(sig)
                    Eliminar (p)
                    p ← head
        De lo contrario
            n ← 1
            Mientras n < pos-1
                n ← n +1
                p ← p(ab)
            q ← p(ab)
            Mientras p <> null
                p(ab) ← q(ab)
                Si p(ab) <> null
                    p(ab(ar)) ← p
                Eliminar (q)
                p ← p(sig)
                Si p <> null entonces
                    q ← p(ab)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_ortogonal::borrar_renglon(int pos)
{
    nodo p,q;
    int n;
    if (head != NULL)
    {
        p = head;
        if (pos <= renglones())
        {
            if (pos == 1)
            {
                q = head->ab;
                if (q != NULL)
                {
                    while (q != NULL)
                    {
                        q->ar = NULL;
                        if (p==head)
                        {
                            head = q;
                        }
                        delete (p);
                        q = q->sig;
                        if ( q != NULL)
                            p = q->ar;
                    }
                }
                else
                {
                    while (p!= NULL)
                    {
                        head = head->sig;
                        delete(p);
                        p = head;
                    }
                }
            }
          else
            {
                n = 1;
                while (n < pos-1)
                {
```

```
                        n++;
                        p = p->ab;
                    }
                    q = p->ab;
                    while (p != NULL)
                    {
                        p->ab = q->ab;
                        if (p->ab != NULL)
                            p->ab->ar = p;
                        delete(q);
                        p = p->sig;
                        if (p != NULL)
                            q = p->ab;
                    }
                }
            }
        }
        else
            cout << "Lista vacía...";
}
```

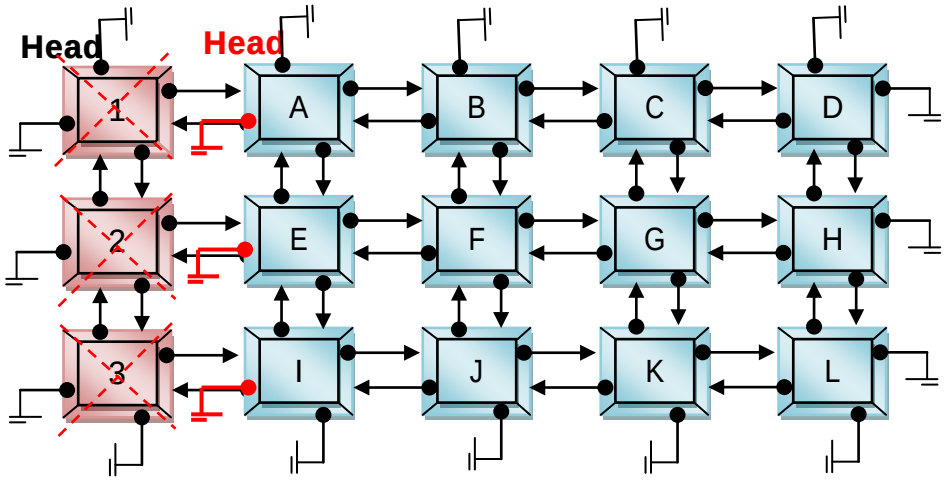
6.17 Eliminar cualquier columna

Diseñar un algoritmo que permita eliminar cualquier columna de una lista ortogonal lineal.

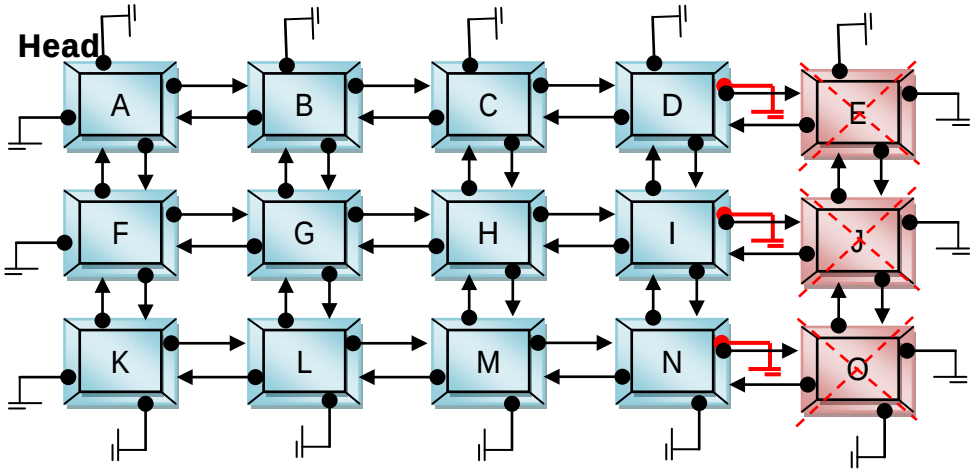
Análisis del problema

Para resolver este problema es necesario conocer y validar la posición de la columna que se eliminará. Se ubica un apuntador en el primer nodo de la lista y se avanza en el primer renglón hasta encontrar la ubicación correcta de la columna, posteriormente se recorre toda la columna para ir eliminando nodo por nodo. Para la solución de este problema se tienen que considerar los casos siguientes:

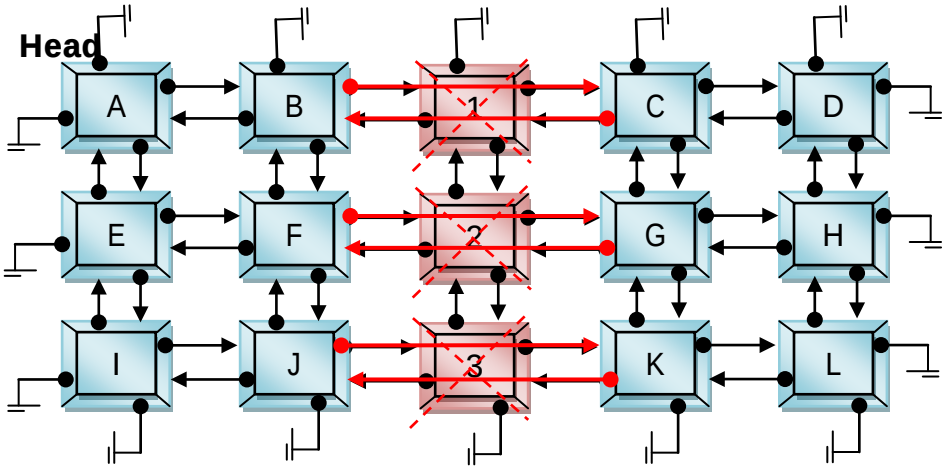
Caso 1: La columna se encuentra ubicada en la primera posición de la lista.



Caso 2: La columna se encuentra ubicada en la última posición de la lista.



Caso 3: La columna se encuentra ubicada en una posición intermedia dentro de la lista.



Solución en pseudocódigo

```
Si head <> null entonces
    p ← head
    Si pos <= columnas entonces
        Si pos = 1 entonces
            q ← head(sig)
            Si q <> null entonces
                Mientras q <> null
                    q(ant) ← null
                    Si p = head entonces
                        head ← q
                    Eliminar (p)
                    q ← q(ab)
                    Si q <> null entonces
                        p ← q(ant)
            De lo contrario
                Mientras p <> null
                    head ← head(ab)
                    Eliminar (p)
                    p ← head
        De lo contrario
            n ← 1
            mientras n < pos-1
                n ← n +1
                p ← p(sig)
            q ← p(sig)
            Mientras p <> null
                p(sig) ← q(sig)
                Si p(sig) <> null
                    p(sig(ant)) ← p
                Eliminar (q)
                p ← p(ab)
                Si p <> null entonces
                    q ← p(sig)
De lo contrario
    Mensaje ('Lista vacía...')
```

Código en C++

```
void lista_ortogonal::borrar_columna(int pos)
{
    nodo p,q;
    int n;
    if (head != NULL)
    {
        p = head;
        if (pos <= columnas())
        {
            if (pos == 1)
            {
                q = head->sig;
                if (q != NULL)
                {
                    while (q != NULL)
                    {
                        q->ant = NULL;
                        if (p==head)
                        {
                            head = q;
                        }
                        delete (p);
                        q = q->ab;
                        if ( q != NULL)
                            p = q->ant;
                    }
                }
                else
                {
                    while (p!= NULL)
                    {
                        head = head->ab;
                        delete(p);
                        p = head;
                    }
                }
            }
            else
            {
                n = 1;
                while (n < pos-1)
                {
```

```
                        n++;
                        p = p->sig;
                    }
                    q = p->sig;
                    while (p != NULL)
                    {
                        p->sig = q->sig;
                        if (p->sig != NULL)
                            p->sig->ant = p;
                        delete(q);
                        p = p->ab;
                        if (p != NULL)
                            q = p->sig;
                    }

                }
            }

        }
        else
            cout << "Lista vacía...";
}
```

[Bibliografía]

[1] GOODRICH M.,TAMASSIA R., *Data Structures and Algorithms in Java*, USA, Wiley, 2008.

[2] MCMILLAN M., Data *Structures and Algorithms Using C#*, USA, Cambridge University Press, 2007.

[3] ALLEN WEISS, M., *Data Structures and Algorithm Analysis in C++,* USA, Addison Wesley, 2006.

[4] JOYANES A. LUIS, *Algoritmos y Estructuras de Datos: Una Perspectiva en C*., España, Mc Graw-Hill, 2005.

[5] MARTINEZ R., QUIROGA, E., *Estructura de Datos*, *Referencia Práctica con Orientación a Objetos*, España, Thompson Learning, 2004.

[6] Drozdek A. *Data Structures and Algorithms in C++*, USA, Course Technology, 2004.

[7] FRANCH X., *Estructura de Datos*, España, Alfaomega Grupo, 2003.

[8] GOODRICH M.,TAMASSIA R., *Estructura de datos y Algoritmos en Java*, México, CECSA, 2003.

[9] MARTINEZ R. *Estructura de Datos*, Thomson International, 2002.

[10] J. SISA A., *Estructura de Datos y Algoritmos*, Colombia, Prentice Hall, 2001.

[11] CAIRÓ O., GUARDATI S., *Estructura de Datos*, México, Mc Graw-Hill, 2001.

[12] ALLEN WEISS, M., *Estructura de Datos y Algoritmos*, España, Addison Wesley, 2000.

[13] JOYANES L., *Programación en C++. Algoritmos, Estructuras de Datos y Objetos*, España, Mc Graw-Hill, 2000.

[14] ALLEN WEISS, M., *Estructuras de Datos en Java,* España, Addison Wesley, 2000.

[15] LOOMIS M.E.S., *Estructura de Datos y Organización de Archivos*, México, Prentice Hall, 1999.

[16] JOYANES L., ZAHONERO I., FDEZ. M., SANCHEZ L., *Estructura de Datos*, España, Mc Graw- Hill, 1999.

[17] Aho A.. *Estructura de Datos y Algoritmos,* España, Addison Wesley, 1999.

[18] FRANCH X., *Estructura de Datos. Especificación, Diseño e Implementación*, Ediciones de la UPC, S.L, España, 1999.

[19] JOYANES L., ZAHONERO I., FERNANDEZ M., SANCHEZ L., *Estructura de Datos: Libro de problemas,* Mc Graw Hill, España, 1999.

[20] HEILEMAN G. *Estructura de Datos, Algoritmos, y Programación Orientada a Objetos*, España, Mc Graw Hill, 1998.

[21] LANGSAM Y., AUGENSTEIN M., TENEBAUM A., *Estructuras de Datos con C y C++*, México, Prentice Hall Hispanoamericano, 1997.

[Contenido]

Se terminó la impresión de Problemas resueltos de listas en agosto de 2012, en el Taller de Artes Gráficas de la UABCS, carretera al sur km 5.5, C.P. 23080. El tiraje sobre papel bond ahuesado de 45 kg, es de quinientos ejemplares. El cuidado de la edición estuvo a cargo de la Lic. Lirio Robles García.