

**Cristina Díez Sobrino**

Tabla de
contenido

[Método intercambiar:](#_Toc2896_WPSOffice_Level1) [3](#_Toc2896_WPSOffice_Level1)

[Método eliminar recursivamente:](#_Toc28534_WPSOffice_Level1) [3](#_Toc28534_WPSOffice_Level1)

[Método eliminar Iterativamente:](#_Toc12947_WPSOffice_Level1) [3](#_Toc12947_WPSOffice_Level1)

[Método ordenada:](#_Toc14874_WPSOffice_Level1) [3](#_Toc14874_WPSOffice_Level1)

[Método mayor:](#_Toc1115_WPSOffice_Level1) [4](#_Toc1115_WPSOffice_Level1)

[Método sumar todos:](#_Toc18632_WPSOffice_Level1) [4](#_Toc18632_WPSOffice_Level1)

[Método invertir:](#_Toc228_WPSOffice_Level1) [4](#_Toc228_WPSOffice_Level1)

[Método insertar posición:](#_Toc15255_WPSOffice_Level1) [4](#_Toc15255_WPSOffice_Level1)

[Método insertar al final:](#_Toc18650_WPSOffice_Level1) [4](#_Toc18650_WPSOffice_Level1)

[Método insertar al principio:](#_Toc30127_WPSOffice_Level1) [5](#_Toc30127_WPSOffice_Level1)

[Método mostrar lista:](#_Toc2885_WPSOffice_Level1) [5](#_Toc2885_WPSOffice_Level1)

## Método intercambiar:

Se pide que el usuario introduzca dos posiciones entre 1 y n (donde n es el número de elementos de la lista) para intercambiar los valores que haya en esas posiciones.

Se guardan en pos1 y pos2.

Se hace un control de errores para que no se pueda seleccionar posiciones fuera del rango.

Se obtienen los valores recorriendo la lista desde el start con un contador que cuenta las posiciones para que coincidan con pos1 o pos2 y guardar los valores de los nodos mediante una variable auxiliar y dos variables tipo nodo(val1 y val2) para los dos valores.

Una vez obtenidos los valores, se intercambian los “info” (siendo info el dato del valor del nodo) de val1 y val2 guardando el valor de una de ellas para que no se pierda en una variable float (value).

**Complejidad**: O(nlogn)

## Método eliminar recursivamente:

Se pasa por parámetro el número que el usuario quiere eliminar. Se controla que la lista no esté vacía y que el número que ha introducido el usuario esté en la lista. Para ello se utiliza el método buscarNodo que devuelve -1 si el número no ha sido encontrado. Si lo encuentra devuelve la primera posición en la que ha sido encontrado. Si hay un solo valor y se borra, la lista se vacía y start y last son NULL.

Si se encuentra se borra “desvinculando” ese nodo de la lista y se resta-1 al contador de elementos:

count--; *//quitar uno al contador de elementos*last->next = s->next;  
s->next->previous = last;  
start = s->next;  
cout << **"Elemento eliminado"** << endl;

Al final se vuelve a ejecutar buscarNodo y si es distinto de -1, es decir, se ha encontrado otra vez el número en la lista porque estaba repetido, se vuelve a ejecutar eliminarRecursiv().

**Complejidad**:O(nlogn)

## Método eliminar Iterativamente:

Es similar a eliminarRecursiv() pero cambia en que este método se ejecutará varias veces por un while que indica que mientras que buscarNodo no devuelva -1 se ejecute el código y al final vuelva a buscar. Cuando buscarNodo devuelva -1, saldrá del bucle while y se dejará de ejecutar.

**Complejidad**: O(n^2)

## Método ordenada:

Este método comprueba si la lista está ordenada.

Se comprueba que la lista no esté vacía. Utilizamos una variable num que la igualamos al primer valor, es decir el valor de start. Hacemos un bucle for que recorra toda la lista utilizando la variable count que cuenta la cantidad de elementos que hay. En la primera vuelta entra en el if porque hemos igualado num a info de start pero después en s se guarda el valor del siguiente nodo.Por lo tanto si num es más pequeño que el valor del nodo que se guarda en s, ordenada será true y si no ordenada será false. Es decir, vamos guardando en num el valor del nodo anterior y con s lo comparamos con el siguiente.

**Complejidad**: O(n^2)

## Método mayor:

Este método guarda el valor más grande en una variable “mayo”. La inicializamos como el valor de “info” de start, para compararla con los demás nodos.

Se recorre la lista con un for y se guarda el valor del nodo en “s”. Si la “info” de s es mayor que mayo se guarda en mayo y se reinicia el contador a 1. Si “info” de s es igual a mayo, se suma 1 al contador, guardando así las veces que aparece el número mayor.

**Complejidad**:O(n)

## Método sumar todos:

Este método guarda los valores en una variable “sum” que se suma con los valores de las anteriores. Se recorre la lista con un for nuevamente y “info” de s se suma a lo que ya había en sum. S pasa a ser el siguiente nodo.

**Complejidad**: O(n)

## Método invertir:

Este método invierte la lista con dos nodo ”p1” y “p2”. P1 se iguala al principio y p2 al siguiente de p1. El siguiente de p1 se pone como NULL y p1 previous coge el valor que habíamos guardaod en p2:

p1 = start;  
p2 = p1->next;  
p1->next = **NULL**;  
p1->previous= p2;

Para ver si hemos acabado de dar la vuelta se utiliza un while que dice que se ejecute el código hasta que p2 sea igual que start y se da la vuelta a los nodos utilizando p2 y almacenando en p1:

p2->previous = p2->next;  
p2->next = p1;  
p1 = p2;  
p2 = p2->previous;

**Complejidad**: O(n)

## Método insertar posición:

Se pide el valor a introducir (v) y la posición en la que se quiere introducir(pos) y se utiliza el metodo getNod que crea un nodo para introducirlo (t). Se comprueba que la lista no esté vacía y si se quiere introducir en la posición 1 , como es el principio se tienen que igualar a start y last al ser una lista doblemente enlazada circular. Si no, se inserta en la posición que nos piden si no está fuera de los límites y se cuenta uno más en el contador de elementos “count”.

**Complejidad**:O(n^2)

## Método insertar al final:

Funciona de manera similar al anterior excepto que el valor introducido se coloca al final de la lista y permite introducir un elemento cuando la lista está vacía. Si start y last son null, la lista está vacía por lo tanto “t” se igualará a last y start para ser el nuevo nodo de la lista y el siguiente y anterior de last y start serán null:

**if** (start == last && start == **NULL**) {  
 cout<<**"Elemento insertado en la lista vacia"**<<endl;  
 start = last = t;  
 start->next= last->next = **NULL**;  
 start->previous = last->previous= **NULL**;

Si la lista no está vacía se introduce al final diciendo que last sea t:

last->next= t;  
t->previous= last;  
last = t;  
start->previous = last;  
last->next= start;

**Complejidad**:O(n)

## Método insertar al principio:

Funciona de manera similar al anterior pero en vez de que t sea last, t se convierte en start:

t->next = start;  
start->previous = t;  
start = t;  
start->previous = last;  
last->next = start;

**Complejidad**:O(n)

## Método mostrar lista:

Si la lista está vacía devuelve un mensaje de “Lista vacia”. Si no, una variable nodo “s” se iguala a start y se recorre la lista con un for imprimiendo el “info” de cada nodo y después igualando “s” al siguiente nodo (s = s->next)