Universidad Autónoma de Nuevo León

Facultad de Ciencias Físico Matemáticas

Tarea 2. Listas

Estructura de Datos

Grupo 032

M. I. A. Ernesto Jesús Solís Valenzuela

2087472 Sebastian Calderon Carrillo

2148314 José Santiago Castro Reyes

San Nicolás de los Garza, Nuevo León Lunes 16 de Febrero de 2026

Índice

1. Introducción

Las estructuras de datos son aquellas que nos permiten almacenar una cantidad de elementos en su contenido, por lo general de un mismo tipo de datos.

En unidades de aprendizaje pasadas, estudiamos el uso de la estructura de datos llamada arreglo. Esta es una estructura estática, es decir, no puede crecer o disminuir su tamaño después de haber sido declarada, debido a que el espacio en memoria se reserva en tiempo de compilación.

En esta tarea, se investigará el uso de una estructura de datos que busca cumplir una función similar, con la diferencia de que se busca su manejo dinámico, o que pueda aumentar y disminuir de tamaño.

La estructura de datos dinámica más elemental es conocida como lista enlazada simple, donde, con el uso de nodos que contienen un dato y un apuntador al siguiente elemento, enlazamos estos nodos entre sí para crear una lista.

En este trabajo, se investigará y explicará el funcionamiento y uso de diferentes modificaciones de estas listas, que expanden su uso más allá de lo que una simple nos podría ayudar.

2. Definición

En este trabajo utilizaremos tres tipos de listas enlazadas diferentes, siendo estas las listas doblemente enlazadas, las listas circulares y las listas circulares doblemente enlazadas. Cada una con su respectivo uso. Empezaremos trabajando con las doblemente enlazadas.

Una lista doblemente enlazada es una estructura de datos dinámica, es decir, que puede crecer de tamaño. Esto es causado por la propiedad de sus elementos, o en este caso sus *nodos.*

Un nodo es una estructura que almacena campos importantes para su funcionamiento, el cual veremos después. Antes, es importante mencionar que estos nodos son dinámicos debido a que se consiguen por medio de la función malloc(), que reserva memoria en el *heap* en tiempo de ejecución. Esto hace posible la creación y eliminación de estos.

Los nodos se conforman por dos o tres campos, usualmente denotados *dato, anterior* y *siguiente.* Esta discrepancia se debe a que no todas las listas poseen el campo de *anterior,* más todas si llevan su campo de *dato* y de *siguiente.*

El campo de *dato* almacena algún valor que se quiera almacenar, pudiendo ser éste de cualquier tipo de dato. Los campos de *siguiente* y *anterior* almacenan en su contenido la dirección de un nodo, siendo estos la dirección del nodo siguiente en la lista y la dirección del nodo anterior respectivamente. Estos campos son de tipo apuntador a estructura *NODO*, cuya declaración veremos en el capítulo de mismo nombre.

Este tipo de listas reciben su nombre debido a que poseen la característica de tener la dirección de ambos nodos tanto enfrente como detrás de sí. En el caso de una lista enlazada simple, utilizando el nodo de cabeza, o el primer nodo, solo

teníamos la opción de movernos hacia enfrente, por lo que era completamente vital no perder la dirección del primer apuntador.

Sin embargo con el uso de listas doblemente enlazadas, es más fácil encontrar el primer y último elemento, debido a que su recorrido puede darse de inicio a fin o viceversa. Por buena práctica, procuraremos mantener siempre bien definida la ubicación del primer elemento.

En comparación, las listas circulares son más parecidas a una lista simple, sin embargo, poseen la peculiaridad de que la dirección del nodo siguiente en el último nodo es el valor del primer nodo, creando así una lista que se le dice circular.

Para finalizar, las listas circulares doblemente enlazadas son listas que poseen ambas características de ser circulares, osea que el apuntador del nodo siguiente en el último elemento apunta al primero, y además poseen las direcciones de los nodos que van detrás, no solo los del frente. Por consiguiente, significa que una lista circular doblemente enlazada posee en primer elemento la dirección del último.

Esto las hace particularmente flexibles a la hora de manejar sus datos, aunque su complejidad puede ser innecesaria para algunos trabajos. A continuación, veremos la declaración de las listas.

3. Declaración

Para declarar una lista enlazada, en este caso en el lenguaje de C, utilizaremos los tres campos que vimos en la definición, siendo estos los de *dato, siguiente* y *anterior*. Para este ejemplo, consideraremos el campo de *dato* como un entero, sin embargo, es importante recalcar que este puede ser de cualquier tipo de dato deseado.

```
typedef struct nodo
{
      int dato;
      struct Nodo *siguiente;
      struct Nodo *anterior; /NOTA, solo se pone este campo en
doblemente enlazadas, sea regular o circular*/
} NODO;
```

Código 1.1. Declaración de un nodo en una lista doblemente enlazada en C.

Utilizamos typedef para facilitar el manejo de los nodos y por consiguiente, de las listas. En vez de escribir *struct nodo ...*, escribiremos *NODO ...* para trabajar con estos.

Para declarar el primer nodo de nuestra lista, que marcará el órden en el resto de la lista, tenemos que crear un apuntador al NODO en el flujo que nos sea más importante, en este caso y en la mayoría, se realizará en *main()*. Esto se realiza de la siguiente manera.

```
int main()
{
. . .
NODO *cabeza = NULL;
. . .
}
```

Código 1.2. Declaración del apuntador cabeza de una lista doble enlazada en C.

Aunque no sea vital, es buena práctica inicializar cualquier apuntador como *NULL*, para asegurarnos que no contenga ninguna dirección "*basura*", o contenido restante en memoria.

Utilizando este apuntador, podemos comenzar con sus operaciones, que son por lo general, la inserción y eliminación. Profundizaremos en estas en un capítulo siguiente.

A continuación, veremos una representación de un nodo de este tipo de listas y de una lista en sí.

4. Representación gráfica

Como sabemos, un nodo contiene dos o tres campos, para la observación práctica, utilizaremos los nodos de tres campos, pertenecientes a las listas doblemente enlazadas como se puede observar en la siguiente figura.

Figura 1.1. Representación de un nodo en una lista doblemente enlazada.

Para poder trabajar con listas, ocupamos solicitar un espacio en memoria y después, con la dirección conseguida y, en caso de que tengamos otros, con las direcciones previas, trabajar en base a ellos.

Supongamos que quisiéramos crear una lista, y por consiguiente un nodo para asignarle el valor de 10, entonces, Nuestro nodo se vería de la siguiente forma.

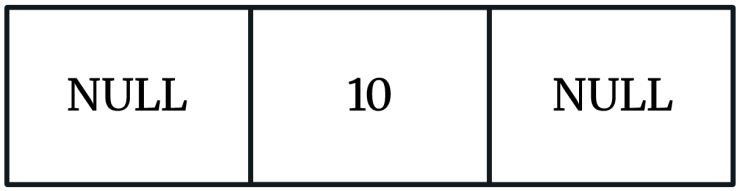


Figura 1.2. Representación de un nodo con un valor de 10.

Al igual que el apuntador cabecera, los apuntadores de *anterior* y *siguiente* se inicializan en NULL para evitar errores.

Como cada nodo es dinámico, cada uno de ellos tiene una dirección en memoria distinta, debido a que esta se obtiene en el momento de ejecución con el uso de la función malloc. En la siguiente figura, se puede observar ese comportamiento con dos nodos. Para este ejemplo, el nodo 1 valdrá 50 y el nodo 2 100.

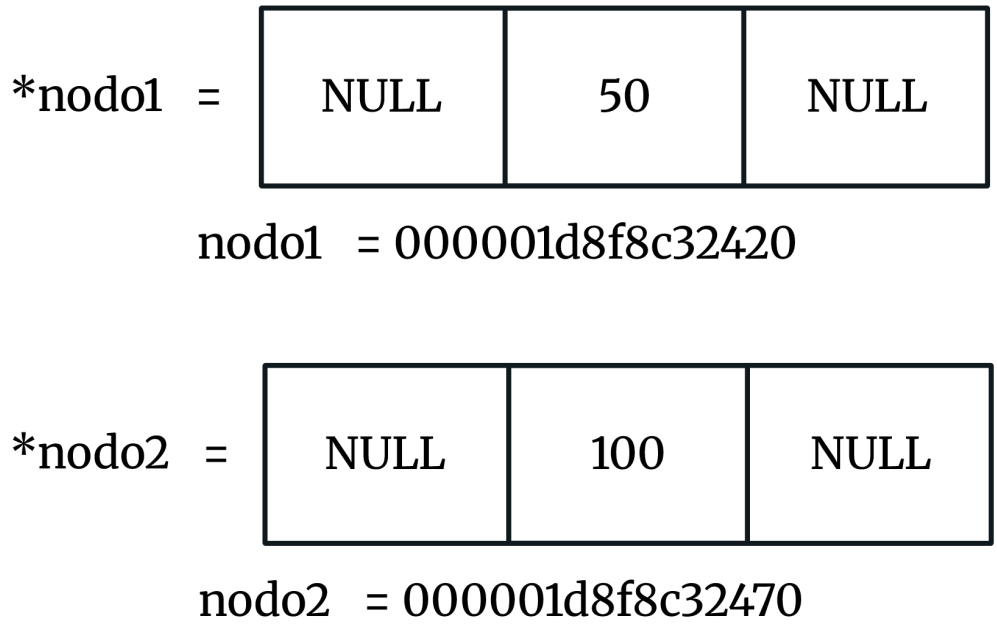


Figura 1.3. Representación de las direcciones de memoria de un nodo.

Para este ejemplo, utilizamos el operador de desreferencia para apuntar al contenido de la dirección que contiene el apuntador, sin embargo, si no se utiliza, podemos ver la dirección que se les asignó en memoria.

En este ejemplo los dos nodos son independientes el uno al otro, sin embargo, si se quisieran enlazar para crear una lista, se debería de asignar la dirección de dicho nodo a algún apuntador del otro nodo. En este caso definiremos a *nodo1* como el primer elemento y *nodo2* como el segundo.

Esto significa que deberemos de asignar el contenido de *nodo1->siguiente* como la dirección de *nodo2*, y el contenido de *nodo2->anterior* como la dirección de *nodo1*. Esto se vería representado de la siguiente manera.

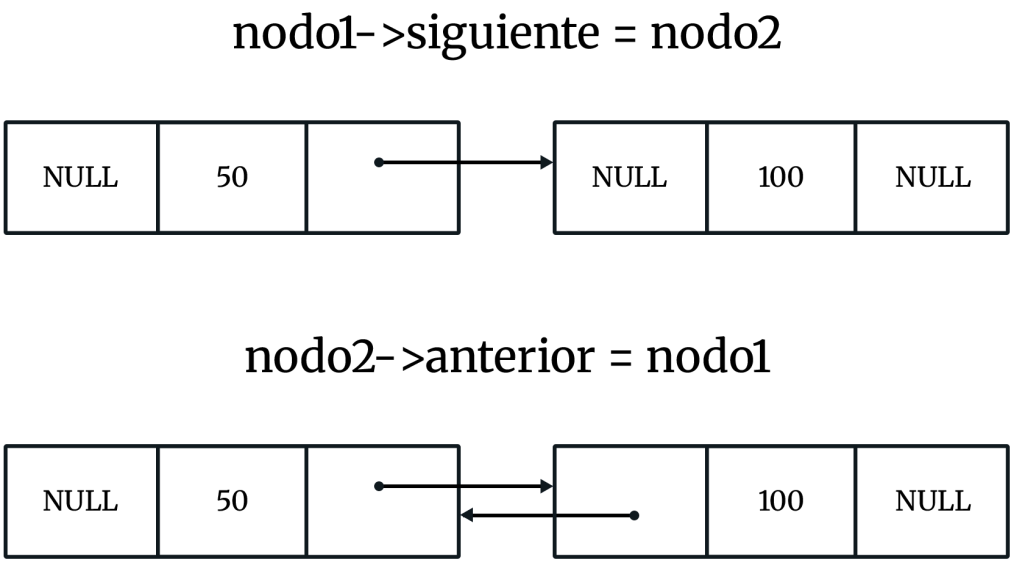


Figura 1.4. Representación del enlace de nodos en una lista enlazada.

En el lenguaje C, se utiliza el operador de flecha para referirnos al elemento de una estructura a través de un apuntador, debido a esto, observamos que nodo1 y nodo2 ambos son apuntadores a una estructura.

Veamos ahora el caso de una lista circular. Como estas no poseen la característica de ser doblemente enlazadas, no es necesario el campo de *anterior*, por lo que un nodo de una lista circular se vería de la siguiente manera.

Figura 1.5. Representación de un nodo en una lista circular.

Como estas listas tienen el apuntador del último elemento apuntando al primer elemento, se suele inicializar apuntando a sí mismo, como podemos ver en la siguiente figura. Consideraremos que el campo de *dato* tiene un valor de 7.

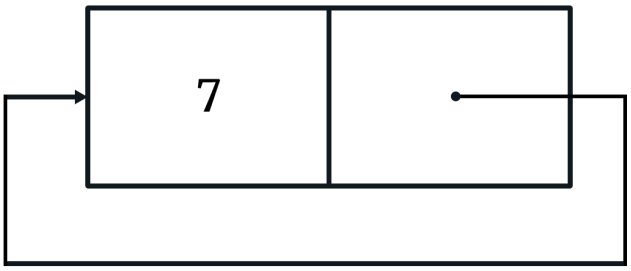


Figura 1.6. Representación del enlace a sí mismo en un nodo en una lista circular.

En caso de que fuesen más elementos, es más sencillo comprender cómo se manejaría este tipo de listas. Veamos por ejemplo el siguiente caso, donde tenemos dos nodos, uno con valor de 30 y el otro de 90.

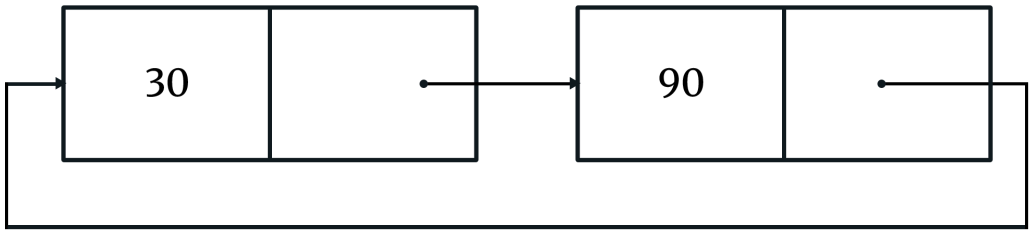
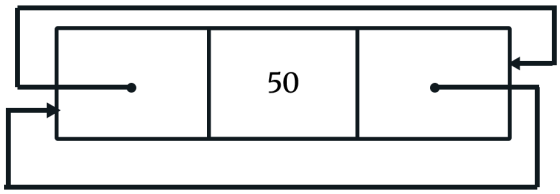


Figura 1.7. Representación del enlace entre nodos en una lista circular.

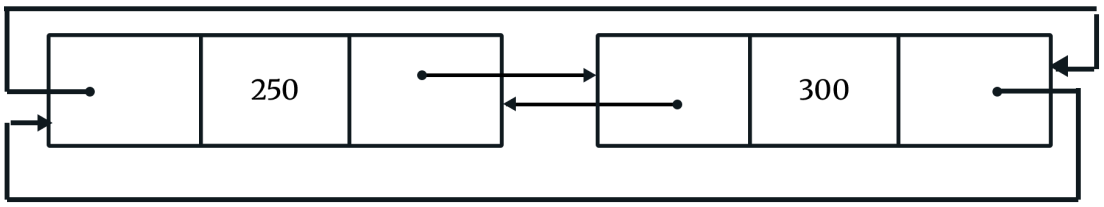
Después, tenemos las listas circulares doblemente enlazadas. Debido a su similitud con los otros tipos de listas, mostraremos directamente la representación de un nodo individual y de una lista circular doblemente



enlazada.

Figura 1.8. Representación del enlace a sí mismo en un nodo de una lista circular doblemente enlazada.

Figura 1.9. Representación del enlace entre nodos en una lista circular doblemente enlazada.



5. Operaciones

Las dos operaciones más comunes para cualquier tipo de lista dinámica son la inserción y eliminación de elementos. Para el caso de nuestro trabajo, la implementación de la inserción de elementos será por orden numérico de menor a mayor, significando que los números más pequeños irán por el inicio y los más grandes al final. La implementación de la eliminación será por medio de una búsqueda, donde se recorrerá la lista en busca del elemento que se desea eliminar para proceder con la acción. En caso de que no se encuentre, no se elimina nada.

Veremos a continuación el algoritmo y diagrama de flujo para la inserción de elementos, seguida de una explicación breve.

5.1. Inserción de un elemento en una lista doblemente encadenada

```
Algoritmo Insertar_Lista

{Este algoritmo inserta un elemento en una lista doblemente encadenada
manteniendo un orden ascendente}


Variables

CABEZA: apuntador a NODO {apuntador al primer elemento}

NUEVO, ACTUAL, PREVIO: apuntadores a NODO

DATO_ENTRADA: Entero


Inicio

{Creación del nodo}

1. Crea(NUEVO)

2. Si (NUEVO = Nulo) entonces

      Retornar
```

```
3. {Fin del condicional del paso 2}


{Asignación de datos}
4. Leer DATO_ENTRADA
5. NUEVO^.Dato ← DATO_ENTRADA
6. NUEVO^.Siguiente ← Nulo
7. NUEVO^.Anterior ← Nulo


{En caso haya una lista vacía}
8. Si (CABEZA = Nulo)
      entonces
            CABEZA ← NUEVO
      sino
            {Búsqueda de posición}
            ACTUAL ← CABEZA
            PREVIO ← Nulo


{Recorremos mientras no sea el final y el dato actual sea menor al
nuevo}
8.1 Mientras ((ACTUAL <> Nulo) Y (ACTUAL^.Dato < DATO_ENTRADA))
Repetir
      PREVIO ← ACTUAL
      ACTUAL ← ACTUAL^.Siguiente
8.2 {Fin del ciclo del paso 8.1}


{Inserción según la posición encontrada}
{En caso de insertar al final}
Si (ACTUAL = Nulo) Entonces
```

```
      PREVIO^.Siguiente ← NUEVO
      NUEVO^.Anterior ← PREVIO
      NUEVO^.Siguiente ← Nulo
{En caso de insertar al inicio}
Sino Si (PREVIO = Nulo)
      Entonces
      NUEVO^.Siguiente ← CABEZA
      CABEZA^.Anterior ← NUEVO
      CABEZA ← NUEVO

{En caso de insertar en el medio}
Sino
      PREVIO^.Siguiente ← NUEVO
      NUEVO^.Anterior ← PREVIO
      NUEVO^.Siguiente ← ACTUAL
      ACTUAL^.Anterior ← NUEVO
Fin_Si
Fin_Si
Fin
```

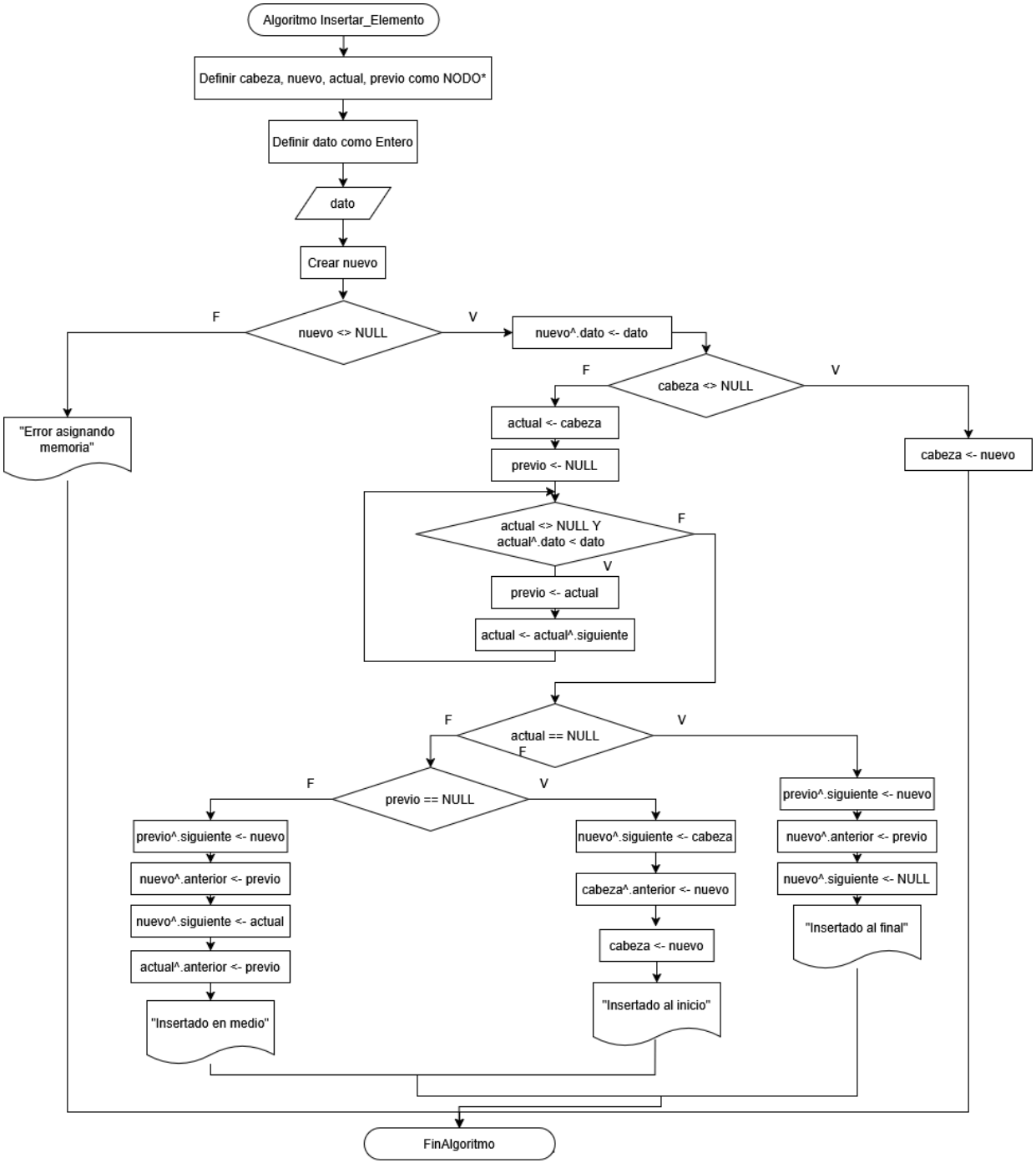


Figura 1.5. Diagrama de flujo de la inserción de un elemento en una lista doblemente encadenada

Para la inserción de un elemento con nuestra implementación, necesitamos de un solo valor antes de la llamada a la función, siendo este el nodo cabeza. Este nodo controlará el inicio de las iteraciones de la lista, y con este, y otros apuntadores de trabajo, se recorrerá la lista en búsqueda de la posición donde se insertará.

Iniciando el proceso, vemos las declaraciones de variables, principalmente las de tipo NODO* y la de tipo entero. Todas las variables de tipo NODO* son apuntadores a la estructura deseada.

Después de la declaración, se lee el valor de nuestro campo *dato*, y se reserva memoria para el nuevo nodo.

Seguido de esto, se hace una validación que evalúa al apuntador *nuevo,* donde, en caso que sea igual a *NULL,* o que no se haya asignado correctamente la memoria. Si es así, se imprime un mensaje de error y termina el flujo de la función, en caso contrario, continuamos con el algoritmo.

Ya habiendo validado el apuntador de cabeza, realizamos otra validación evaluando al puntero cabeza, ya que si fuese *NULL*, significa que la lista no contiene elementos, por lo que el nuevo elemento sería el primer elemento de la lista, y no tiene que iterar a través de ella.

En caso de que la lista sea diferente de *NULL*, creamos un ciclo para recorrer la lista. Mientras *previo* sea diferente de *NULL*, significa que no se recorrió la lista, entonces se posiciona al inicio. Si *actual* es igual a *NULL*, significa que se recorrió la lista y se llegó al final, por lo que se inserta el elemento al final. Caso contrario, se inserta el elemento en el medio siguiendo el orden numérico.

5.2. Eliminación de un elemento en una lista doblemente encadenada

```
Algoritmo Eliminar_Lista

{Este algoritmo busca y elimina un nodo específico en una lista
doblemente encadenada}


Variables

      CABEZA: apuntador a NODO {apuntador al primer elemento}

      ACTUAL, PREVIO: apuntadores a NODO

      DATO_BUSCAR: Entero


Inicio

      {Verificar si hay lista}

      1. Si (CABEZA = Nulo) Entonces

            Retornar

      2. {Fin del condicional paso 1}


      3. Leer DATO_BUSCAR

      4. ACTUAL ← CABEZA

      5. PREVIO ← Nulo


      {Recorremos mientras no sea el final y no encontremos el dato}

      6. Mientras ((ACTUAL <> Nulo) Y (ACTUAL^.Dato <> DATO_BUSCAR))
Repetir

            PREVIO ← ACTUAL

            ACTUAL ← ACTUAL^.Siguiente

      7. {Fin del ciclo paso 6}
```

```
      {Verificar resultado de búsqueda}
      8. Si (ACTUAL = Nulo) Entonces
            Escribir "No se encontró nodo con el valor ingresado."
      Sino

            {Caso: El nodo es el primero (PREVIO es nulo)}
            9. Si (PREVIO = Nulo) Entonces
                  {Caso: Es el único nodo}
                  Si (ACTUAL^.Siguiente = Nulo) Entonces
                        CABEZA ← Nulo
                  Sino
                        {Hay más nodos, movemos la cabeza}
                        CABEZA ← ACTUAL^.Siguiente
                        CABEZA^.Anterior ← Nulo
                  Fin_Si

            {Caso: El nodo está en medio o final}
            Sino
                  PREVIO^.Siguiente ← ACTUAL^.Siguiente

                  {Si no es el último nodo, reconectamos el enlace
anterior del siguiente}
                  Si (ACTUAL^.Siguiente <> Nulo) Entonces
                        (ACTUAL^.Siguiente)^.Anterior ← PREVIO
                  Fin_Si
            Fin_Si
```

```
            {Liberación de memoria}
            10. Libera(ACTUAL)
            Escribir "Valor eliminado correctamente."
      Fin_Si
Fin
```

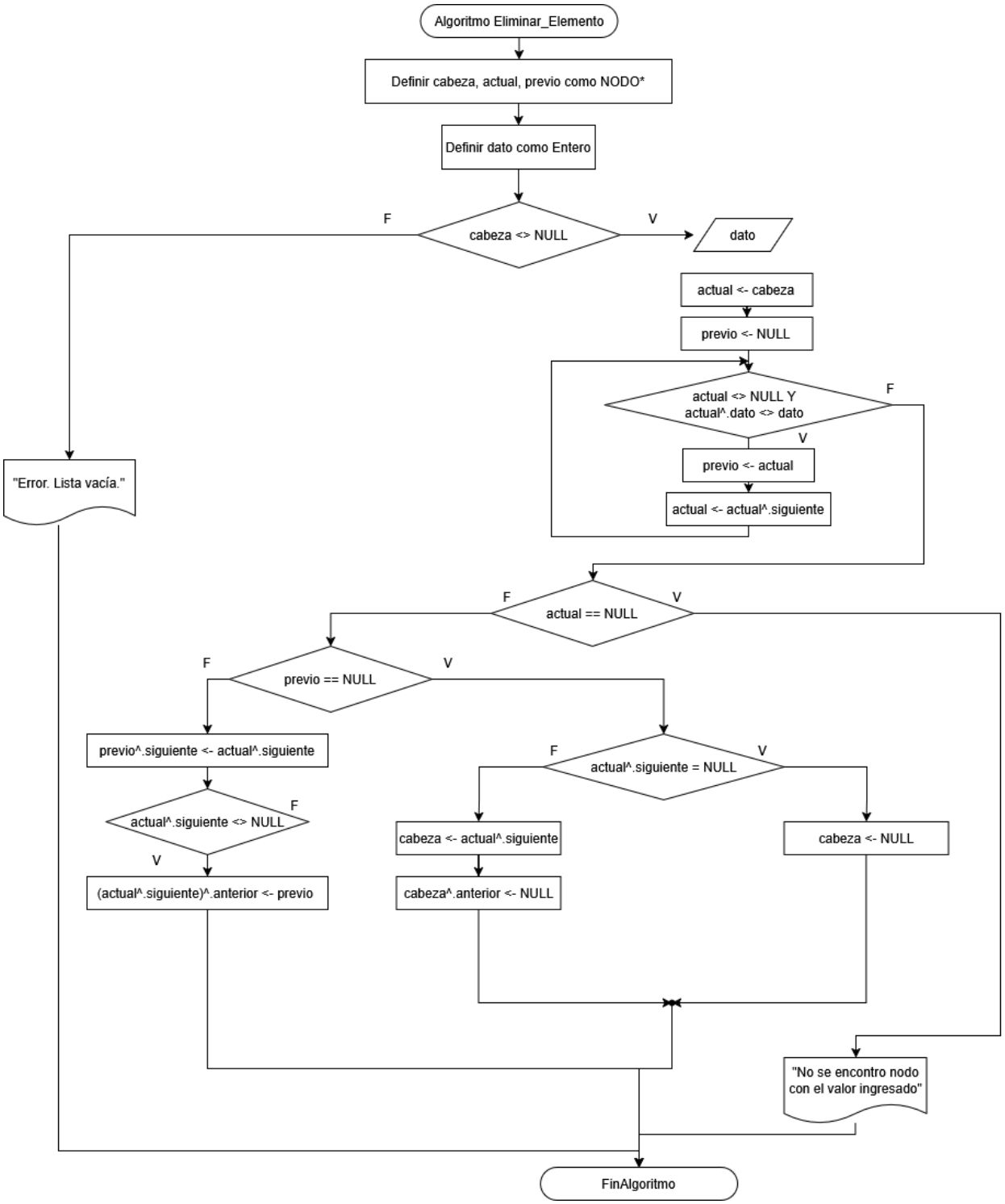


Figura 1.6. Diagrama de flujo de la eliminación de un elemento en una lista doblemente encadenada

En el caso de la eliminación, de igual manera que en la inserción, por la naturaleza de nuestra lista, se tiene que recorrer la lista enlazada para poder hallar (o no) el elemento que queramos eliminar. Dependiendo del tipo de estructura es que utilizamos diferentes apuntadores temporales o de trabajo para poder recorrer la lista encadenada, pero, de manera general, podemos resumir los 3 casos de la siguiente manera: se utilizan dos apuntadores, uno que llamaremos ACTUAL y el cual posicionamos en la dirección a la que apunta la cabecera de la lista. El otro apuntador será PREVIO, que nos ayudará a estar siempre una dirección detrás de ACTUAL para poder acceder al nodo anterior para realizar la eliminación que más adelante se explicara.

Dependiendo del primer valor de ACTUAL podemos ya concluir cierta información: si es NULL, significa que la lista no contiene ningún elemento, de lo contrario, tendrá uno o más elementos. Si este es el caso, se recorre la lista, en caso de que el valor sea hallado, se borra mediante enlazar el nodo PREVIO con ACTUAL->siguiente, haciendo básicamente que se salte el nodo a eliminar, y como tenemos guardada la dirección de ACTUAL, solo es cuestión de hacer free(ACTUAL).

5.3. Inserción de un elemento en una lista circular

```
Algoritmo Insertar_Lista_Circular
{Este algoritmo inserta un elemento en una lista circular simple
manteniendo orden ascendente}


Variables
     CABEZA: apuntador a NODO
     NUEVO, ACTUAL, PREVIO, ULTIMO: apuntadores a NODO
```

```
      DATO_ENTRADA: Entero

Inicio
      {Creación del nodo}
      1. Crea(NUEVO)
      2. Si (NUEVO = Nulo) Entonces
            Retornar
      3. {Fin condicional memoria}

      {Asignación de datos}
      4. Leer DATO_ENTRADA
      5. NUEVO^.Dato ← DATO_ENTRADA
      6. NUEVO^.Siguiente ← Nulo

      {Caso: Lista Vacia}
      7. Si (CABEZA = Nulo) Entonces
            CABEZA ← NUEVO
            NUEVO^.Siguiente ← NUEVO
            Escribir "Nueva lista creada."
      Sino
            {Inicialización punteros}
            ACTUAL ← CABEZA
            ULTIMO ← CABEZA
            PREVIO ← Nulo

            {Ciclo de búsqueda de posición}
            8. Repetir
```

```
                Si (ACTUAL^.Dato ⩽ NUEVO^.Dato) Entonces
                    PREVIO ← ACTUAL
                    ACTUAL ← ACTUAL^.Siguiente
                Sino
                    Interrumpir {Romper ciclo}
                Fin_Si
            Hasta Que (ACTUAL = CABEZA)

            {Caso de insertar al inicio}
            9. Si ((ACTUAL = CABEZA) Y (NUEVO^.Dato ⩽ CABEZA^.Dato))
Entonces
                {Busca el último nodo para cerrar el círculo}
                Mientras (ULTIMO^.Siguiente <> CABEZA) Repetir
                    ULTIMO ← ULTIMO^.Siguiente
                Fin_Mientras

                NUEVO^.Siguiente ← CABEZA
                ULTIMO^.Siguiente ← NUEVO
                CABEZA ← NUEVO
            {Caso de insertar en medio o final}
            Sino
                PREVIO^.Siguiente ← NUEVO
                NUEVO^.Siguiente ← ACTUAL
            Fin_Si

        Fin_Si
Fin
```

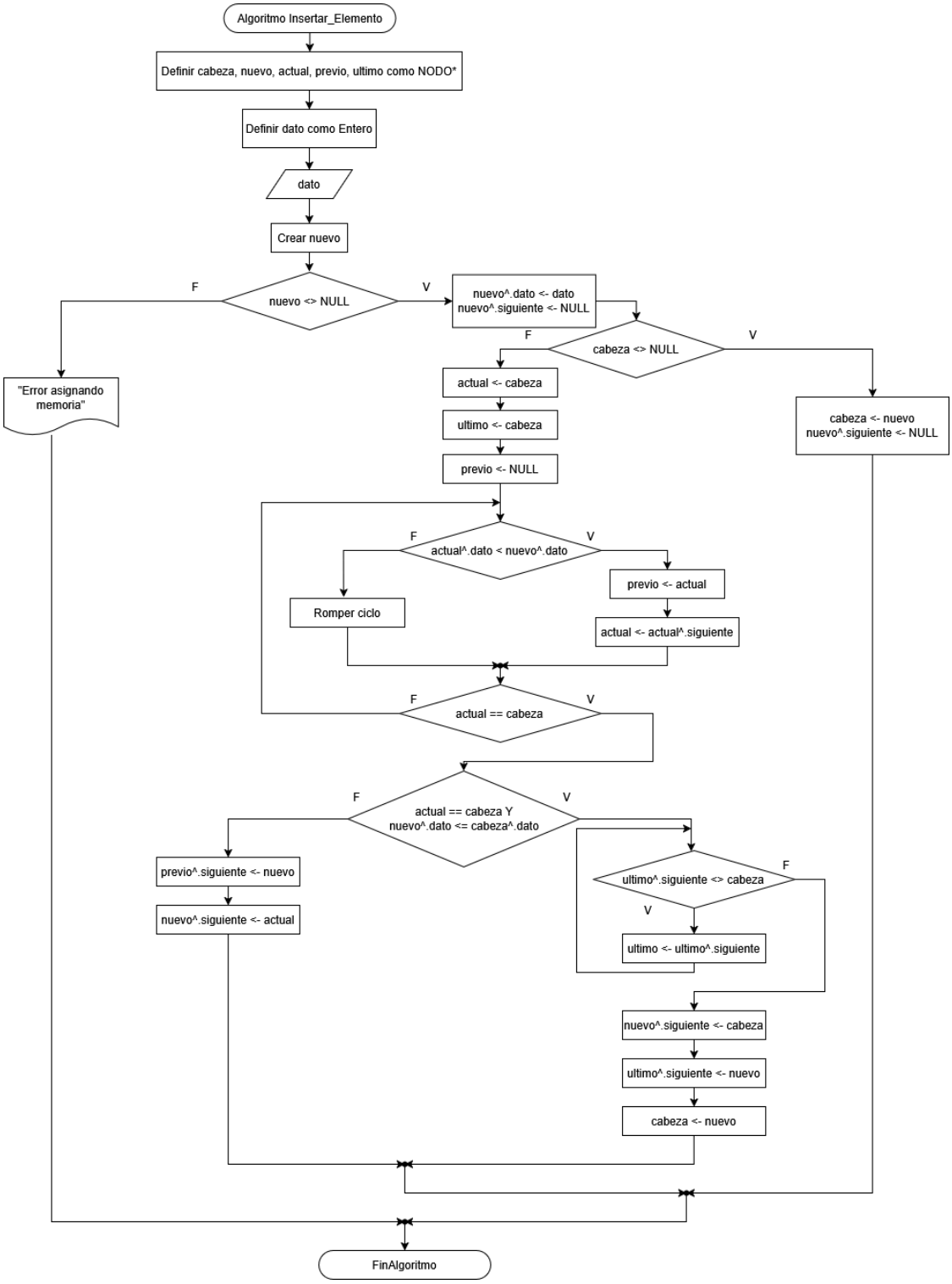


Figura 1.7. Diagrama de flujo de la inserción de un elemento en una lista circular

5.4. Eliminación de un elemento en una lista circular

```
Algoritmo Eliminar_Lista_Circular

{Este algoritmo elimina un nodo en una lista circular simple
reconectando el ciclo}


Variables

      CABEZA: apuntador a NODO

      ACTUAL, PREVIO, TEMP: apuntadores a NODO

      DATO_BUSCAR: Entero


Inicio

      1. Si (CABEZA = Nulo) Entonces

            Escribir "ERROR. No existen nodos."

            Retornar

      2. {Fin del condicional del paso 1}

      3. Leer DATO_BUSCAR

      ACTUAL ← CABEZA

      PREVIO ← CABEZA

      TEMP ← Nulo


      {Búsqueda del nodo}

      4. Repetir

            Si (ACTUAL^.Dato = DATO_BUSCAR) Entonces

                  TEMP ← ACTUAL

                  ACTUAL ← ACTUAL^.Siguiente

                  Interrumpir

            Fin_Si
```

```
      ACTUAL ← ACTUAL^.Siguiente
Hasta Que (ACTUAL = CABEZA)

5. Si (TEMP = Nulo) Entonces
      Escribir "No se encontró el nodo."
Sino
      {Caso de que el nodo a borrar es la cabeza}
      6. Si (TEMP = CABEZA) Entonces

            {Caso: Único nodo en la lista}
            Si (ACTUAL = CABEZA) Entonces
                  CABEZA ← Nulo
                  Escribir "Lista vaciada."

            {Caso de que haya más nodos, la cabeza cambia}
            Sino
                  {Buscar el último para reconectar}
                  Mientras (PREVIO^.Siguiente <> CABEZA) Repetir
                        PREVIO ← PREVIO^.Siguiente
                  Fin_Mientras

                  PREVIO^.Siguiente ← ACTUAL
                  CABEZA ← ACTUAL
            Fin_Si

      {Caso del nodo en medio o final}
      Sino
```

```
                {Buscar el nodo previo al que se va a borrar}
                Mientras (PREVIO^.Siguiente <> TEMP) Repetir
                      PREVIO ← PREVIO^.Siguiente
                Fin_Mientras

                PREVIO^.Siguiente ← ACTUAL
          Fin_Si

          7. Libera(TEMP)
          Escribir "Valor eliminado correctamente."
     Fin_Si
Fin
```

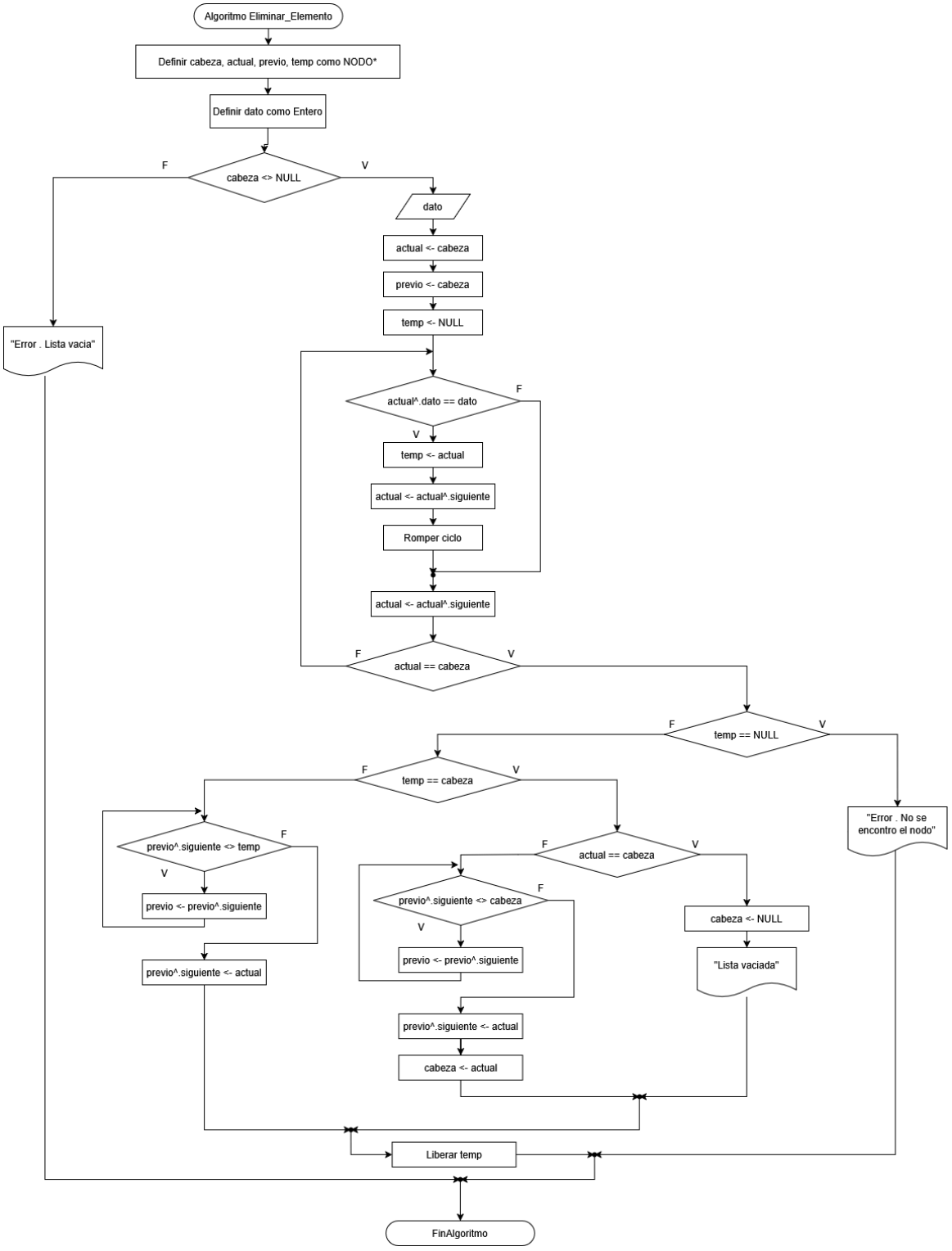


Figura 1.8. Diagrama de flujo de la eliminación de un elemento en una lista circular

5.5. Inserción de un elemento en una lista circular doblemente encadenada

```
Algoritmo Insertar_Lista_Circular_Doblemente_Encadenada

{Este algoritmo inserta un elemento en una lista circular doblemente
encadenada}


Variables

      CABEZA: apuntador a NODO

      NUEVO, ACTUAL, PREVIO, ULTIMO: apuntadores a NODO

      DATO_ENTRADA: Entero


Inicio

      {Creación del nodo}

      1. Crea(NUEVO)

      2. Si (NUEVO = Nulo) Entonces Retornar


      {Asignación de datos}

      3. Leer DATO_ENTRADA

      4. NUEVO^.Dato ← DATO_ENTRADA

      5. NUEVO^.Anterior ← Nulo

      6. NUEVO^.Siguiente ← Nulo


      {Caso de la lista Vacía}

      7. Si (CABEZA = Nulo) Entonces

            CABEZA ← NUEVO

            NUEVO^.Siguiente ← NUEVO

            NUEVO^.Anterior ← NUEVO

            Escribir "Nueva lista creada."
```

```
      Sino
            ACTUAL ← CABEZA

            {Búsqueda de posición}
            8. Repetir
                  Si (ACTUAL^.Dato < NUEVO^.Dato) Entonces
                        PREVIO ← ACTUAL
                        ACTUAL ← ACTUAL^.Siguiente
                  Sino
                        Interrumpir
                  Fin_Si
            Hasta Que (ACTUAL = CABEZA)

            {Caso de insertar al inicio}
            9. Si ((ACTUAL = CABEZA) Y (NUEVO^.Dato ⩽ CABEZA^.Dato))
Entonces
                  ÚLTIMO ← CABEZA^.Anterior

                  NUEVO^.Siguiente ← CABEZA
                  NUEVO^.Anterior ← ULTIMO
                  ULTIMO^.Siguiente ← NUEVO
                  CABEZA^.Anterior ← NUEVO
                  CABEZA ← NUEVO

            {Caso de insertar en medio o en el final}
            Sino
                  PREVIO ← ACTUAL^.Anterior
```

```
                NUEVO^.Siguiente ← ACTUAL
                NUEVO^.Anterior ← PREVIO
                PREVIO^.Siguiente ← NUEVO
                ACTUAL^.Anterior ← NUEVO
            Fin_Si

            Escribir "Se ingresó correctamente el nodo."
      Fin_Si
Fin
```

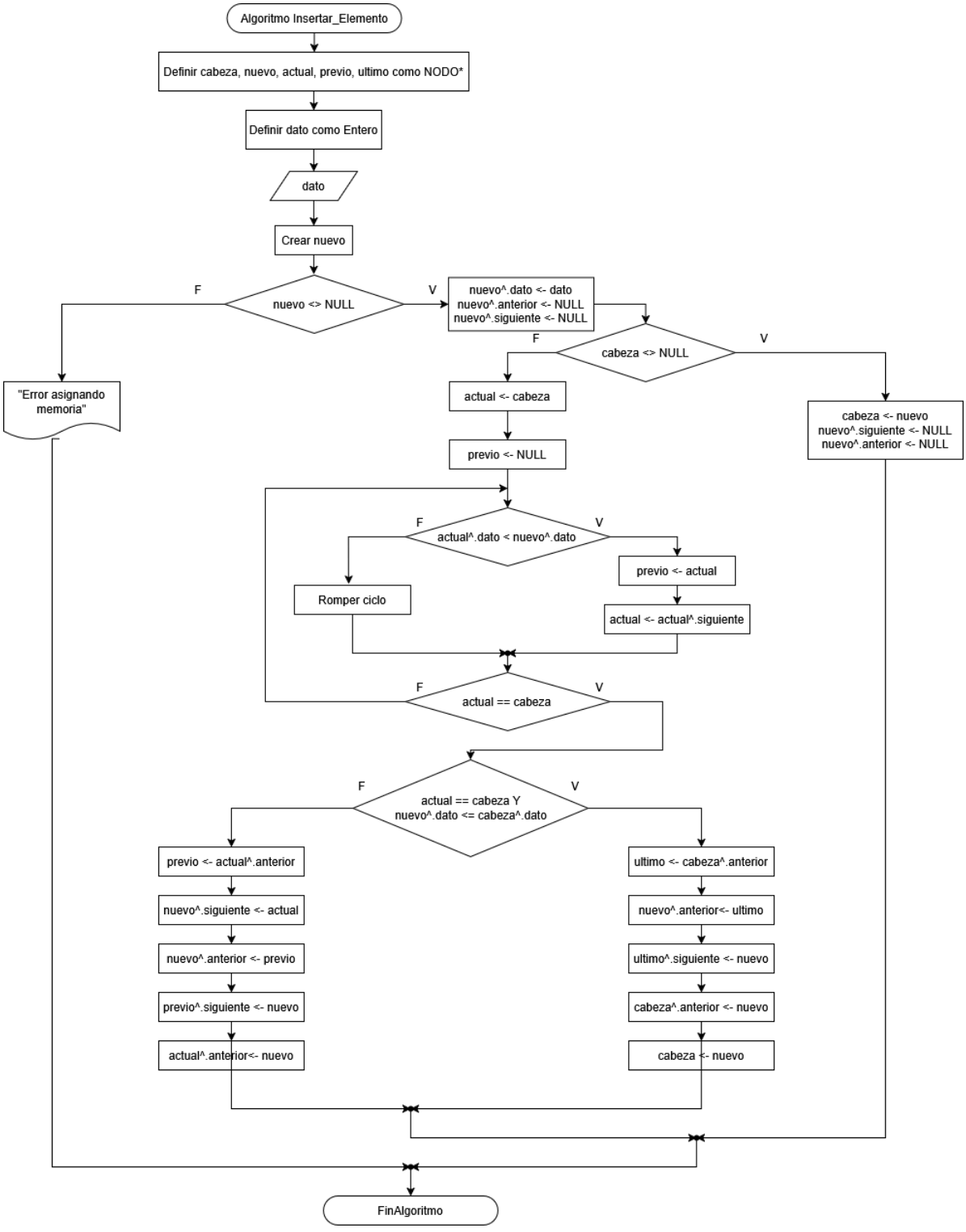


Figura 1.9. Diagrama de flujo de la inserción de un elemento en una lista circular doblemente encadenada

5.6. Eliminación de un elemento en una lista circular doblemente encadenada

```
Algoritmo Eliminar_Lista_Circular_Doblemente_Encadenada

{Este algoritmo elimina un nodo manteniendo la doble ligadura y la
circularidad}


Variables

      CABEZA: apuntador a NODO

      ACTUAL, PREVIO: apuntadores a NODO

      DATO_BUSCAR: Entero


Inicio

      1. Si (CABEZA = Nulo) Entonces Retornar


      2. Leer DATO_BUSCAR

      ACTUAL ← CABEZA

      PREVIO ← Nulo


      {Búsqueda del nodo}

      3. Repetir

            Si (ACTUAL^.Dato = DATO_BUSCAR) Entonces

                  PREVIO ← ACTUAL^.Anterior

                  Interrumpir

            Fin_Si

            ACTUAL ← ACTUAL^.Siguiente

      Hasta Que (ACTUAL = CABEZA)


      {Verificación}
```

```
      4. Si (PREVIO = Nulo) Entonces
            Escribir "No se encontró nodo."
      Sino
            {Caso del nodo encontrado es cabeza}
            5. Si (PREVIO = CABEZA^.Anterior) Entonces

                  {Caso de un único nodo}
                  Si (ACTUAL = CABEZA^.Anterior) Entonces
                        CABEZA ← Nulo
                        Escribir "Lista vaciada."

                  {Caso de la cabeza con más nodos}
                  Sino
                        (ACTUAL^.Anterior)^.Siguiente ←
ACTUAL^.Siguiente
                        (ACTUAL^.Siguiente)^.Anterior ← ACTUAL^.Anterior
                        CABEZA ← ACTUAL^.Siguiente
                  Fin_Si

            {Caso de en medio o en el final}
            Sino
                  PREVIO^.Siguiente ← ACTUAL^.Siguiente
                  (ACTUAL^.Siguiente)^.Anterior ← PREVIO
            Fin_Si

            6. Libera(ACTUAL)
            Escribir "Valor eliminado correctamente."
```

```
    Fin_Si
Fin
```

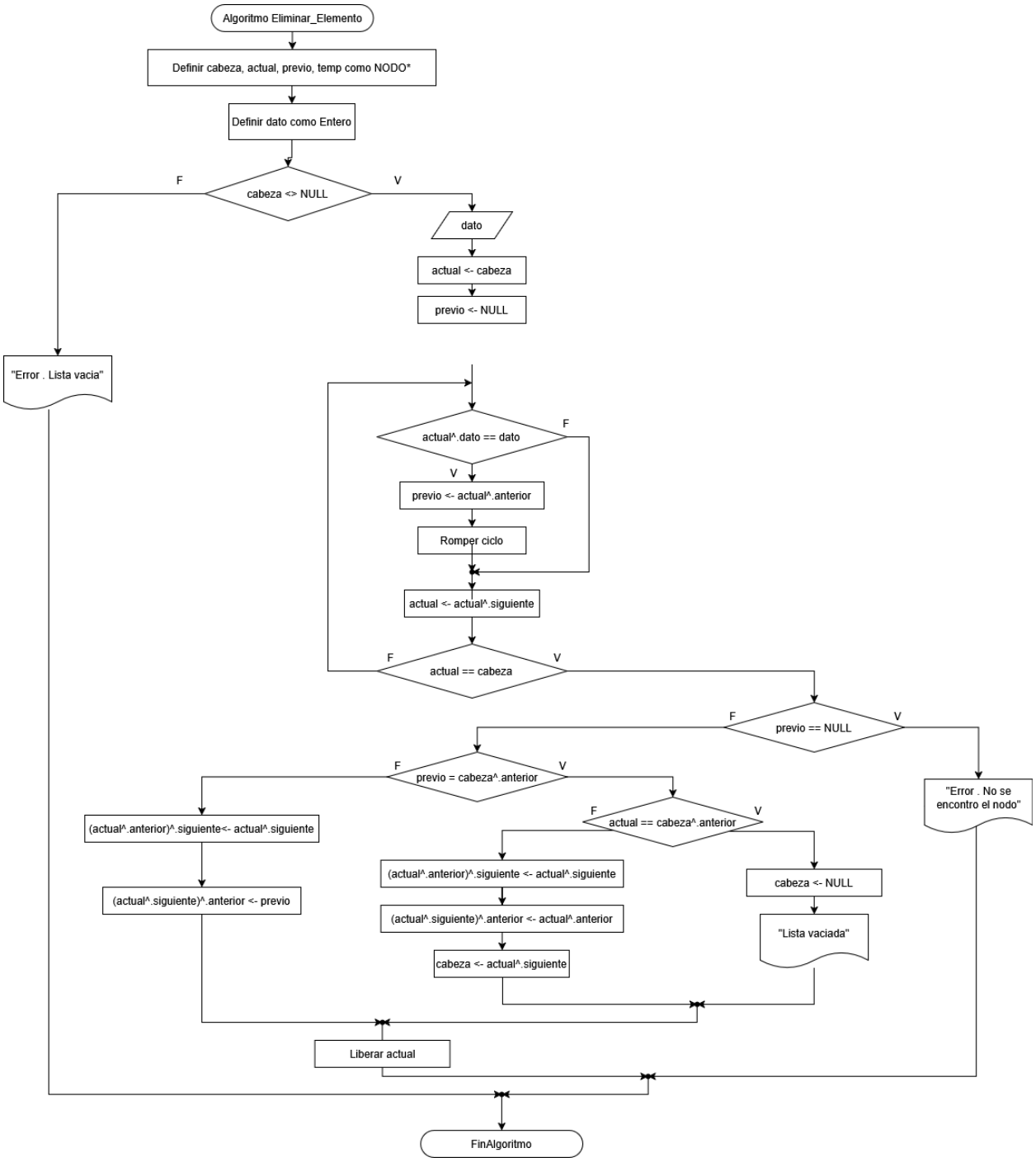


Figura 1.10. Diagrama de flujo de la eliminación de un elemento en una lista circular doblemente encadenada

6. Conclusiones

En la realización de esta tarea pudimos darnos cuenta de las diferentes posibilidades de organizar información con las “pocas” herramientas que proporciona C, teniendo que ingeniar la manera en que podemos optimizar memoria, acomodar información, recorrerla, reorganizarla, etc. Consideramos que este tipo de programación despierta un sentido de la lógica y resolución de problemas, siendo que en otros lenguajes se da por hecho que las estructuras de datos ya están ahí listas para utilizarse.

Por otro lado, aprendimos más sobre el trabajo en equipo y lo que una buena planificación y asignación de tareas puede llegar a lograr. Una de las herramientas principales que utilizamos fue Git, el cual nos ayudó a un correcto flujo de trabajo.

7. Bibliografías

Joyanes Aguilar, L., & Zahonero Martinez, I. (2001). *PROGRAMACIÓN EN C: Metodología, algoritmos y estructura de datos* (Segunda edición). ISBN: 8448130138. Clasificación CDD: 001.6424 J88j 21.

Cairó, O., & Guardati, S. (2006). *Estructuras de datos* (3a. edición). ISBN: 9701059085. Clasificación CDD: 005.73 C123 21.