



## Carátula para entrega de prácticas

Facultad de Ingeniería

Laboratorio de docencia

# Laboratorios de computación salas A y B

*Profesor:*

Edgar Tista García

*Asignatura:*

Estructuras de Datos y Algoritmos I

*Grupo:*

1

*No. de Práctica(s):*

7 y 8

*Integrante(s):*

Ugalde Velasco Armando

*No. de Equipo de  
cómputo empleado:*

*No. de Lista:*

38

*Semestre:*

2020-2

*Fecha de entrega:*

12 de junio de 2020

*Observaciones:*

**CALIFICACIÓN:** \_\_\_\_\_

## OBJETIVOS

**Práctica 7:** Revisar las definiciones, características, procedimientos y ejemplos de las estructuras lineales Lista simple y Lista circular, con la finalidad de comprender sus estructuras e implementarlas.

**Práctica 8:** Revisar las definiciones, características, procedimientos y ejemplos de las estructuras lineales lista doblemente ligada y lista doblemente ligada circular, con la finalidad de comprender sus estructuras e implementarlas.

## ACTIVIDAD 1

a) Elabora un programa para crear una lista de elementos y verificar las funciones de la lista, en este programa el usuario podrá ver un menú con esas opciones.

- I) AGREGAR AL PRINCIPIO
- II) AGREGAR AL FINAL
- III) AGREGAR I-ÉSIMO (AGREGA EN LA POSICIÓN QUE EL USUARIO QUIERE)
- IV) ELIMINAR AL INICIO
- V) ELIMINAR AL FINAL

Se realizó la implementación del menú con las opciones solicitadas, utilizando las funciones de la biblioteca *lista.h*: **addPrincipioLista**, **addFinalLista**, **addlesimoLista**, **borrarPrimero** y **borrarUltimo**, respectivamente. Además, se añadió la opción **salir** del menú.

```
***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Agregar i-esimo
4) Eliminar al inicio
5) Eliminar al final
6) Salir
```

MENÚ PRINCIPAL

El funcionamiento de la biblioteca fue el esperado, sin embargo, la función **addlesimoLista** no contaba con soporte para casos límite. Debido a lo anterior, ésta fue modificada de la siguiente forma:

### CASO LÍMITE 1: LISTA VACÍA

```
// Corner case: lista vacia
if (!lista->head) {
    printf("Se insertara al principio de la lista, ya que esta vacia.\n");
    lista->head = nuevoNodo;
    nuevoNodo->next = NULL;
    return;
```

*Si la lista se encontraba vacía al momento de la inserción, simplemente se insertaba el nuevo nodo a la lista, provocando que éste fuera el nuevo Head.*

### CASO LÍMITE 2: INSERTAR EN POSICIÓN 0

```
// Corner case: insertar al principio (posicion 0)
if (posicion == 0)
{
    lista->head = nuevoNodo;
    nuevoNodo->next = temp;
    return;|
```

*En este caso, únicamente se insertó el nodo al inicio de la lista. No se utilizó la función existente, ya que en este punto del código ya se contaba con el nuevo nodo almacenado.*

### CASO LÍMITE 3: POSICIÓN SOLICITADA FUERA DEL RANGO

```
for(contador=0;contador<posicion-1;contador++)    {
    // Corner case: la posicion solicitada se encuentra fuera del rango
    if(!temp->next)
    {
        printf("Se insertara al final de la lista.\n");
        temp->next = nuevoNodo;
        nuevoNodo->next = NULL;
        return;
    }
    temp=temp->next;
```

*Este caso se presenta al recorrer la lista. Si el usuario proporciona un valor de posición mayor al tamaño de la lista, no es posible insertar el nodo en la posición deseada, por lo tanto, se inserta al final de ésta. Cabe mencionar que también se añadió el manejo de números de posición negativos: si el usuario introduce uno de estos, se le vuelve a solicitar hasta que introduzca un dato válido.*

Se utilizó la convención de las posiciones basadas en el índice 0, es decir, la primera posición tiene índice 0, la segunda 1, y así sucesivamente. Por lo tanto, el elemento se inserta en su respectiva posición (respecto a este criterio).

**b) Agrega a la biblioteca y al menú, la función “buscar” que devuelva como resultado la posición donde se encuentra el elemento. Para este ejercicio se tomará la convención de que el primer elemento ocupará la posición 0.**

La función se implementó de la siguiente forma:

Primero, se comprueba si la lista está vacía. De ser así, se retorna el valor -1 para indicar que el elemento deseado no se encuentra en la lista. Posteriormente, se crea la variable auxiliar **trav** para recorrer la lista, y se le asigna el nodo **head**. También se declara la variable **pos**, que contendrá la posición del nodo **trav**. Se inicializa a cero, ya que, como se solicitó en el ejercicio, la posición del nodo **head** es ésta.

```
int buscarElemento(Lista *lista, int elemento)
{
    if (!lista->head)
    {
        printf("No se encontro el elemento (lista vacia) \n");
        return -1;
    }

    Nodo *trav = lista->head;
    int pos = 0;
```

#### CONSIDERACIÓN DEL CASO LÍMITE DE LISTA VACÍA

Después, utilizando la variable auxiliar mencionada, se recorre la lista en busca del elemento. En cada iteración, si el valor a buscar se encuentra en el nodo **trav**, se retorna su posición (almacenada en **pos**), de lo contrario, se actualiza el nodo **trav** y la variable **pos** al siguiente nodo y su posición, respectivamente.

Si el ciclo finaliza su ejecución (es decir, si se “recorre” toda la lista), entonces la búsqueda no fue exitosa, por lo tanto, se devuelve el valor -1, al igual que en el caso de una lista vacía.

```

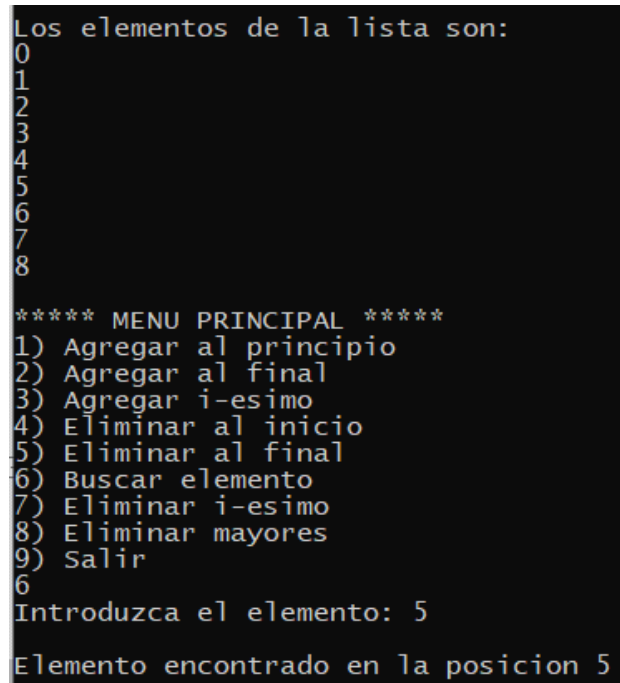
while(trav)
{
    if (trav->val == elemento)
    {
        printf("\nElemento encontrado en la posicion %d\n", pos);
        return pos;
    }
    pos++;
    trav = trav->next;
}

printf("No se encontro el elemento\n");
return -1;

```

#### RECORRIDO DE LA LISTA Y DETERMINACIÓN DEL RESULTADO DE LA BÚSQUEDA

A continuación, se muestra una captura de pantalla de la salida del programa realizado para comprobar el comportamiento correcto de la función:



```

Los elementos de la lista son:
0
1
2
3
4
5
6
7
8

***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Agregar i-esimo
4) Eliminar al inicio
5) Eliminar al final
6) Buscar elemento
7) Eliminar i-esimo
8) Eliminar mayores
9) Salir
6
Introduzca el elemento: 5
Elemento encontrado en la posicion 5

```

SALIDA DEL PROGRAMA

c) Agrega a la biblioteca y al menú, la función “eliminar n-ésimo” la cual eliminará el elemento en la posición que el usuario determine y comprueba el uso de la biblioteca en el programa anterior.

Esta función se implementó de una forma muy similar a la función **addlesimoLista**.

Primero, se solicitó la posición a eliminar al usuario. Es importante mencionar que, al igual que en la función mencionada, se utilizó la convención de índices basados en cero.

```

void eliminarEnesimo(Lista *lista)
{
    int i, posicion;
    printf("Ingrese la posicion: ");
    scanf("%d",&posicion);

    while (posicion < 0)
    {
        printf("Inserte una posicion valida: ");
        scanf("%d", &posicion);
    }
}

```

**SE SOLICITA LA POSICIÓN DEL ELEMENTO A ELIMINAR**

Posteriormente, se consideró el caso límite de una lista vacía: no es posible eliminar ningún elemento, por lo tanto, la función finaliza su ejecución. Si este caso no se presenta, sí es posible eliminar algún elemento, por lo tanto, se procede a crear un nodo temporal para recorrer la lista.

```

// Corner case: lista vacia
if (!lista->head) {
    printf("No se puede eliminar (lista vacia)\n");
    return;
}

Nodo *temp; //creacion de nodo temporal para recorrer
temp=list->head;

```

**CASO LÍMITE: LISTA VACÍA. CREACIÓN DE NODO TEMPORAL**

Después, se considero otro caso límite: eliminar el elemento en la posición 0. Al ya encontrarse definida esta operación (**borrarPrimero**), únicamente se utilizó la función correspondiente para cumplir con la tarea.

```

// Corner case: eliminar al principio (posicion 0)
if (posicion == 0)
{
    borrarPrimero(lista);
    return;
}

```

**CASO LÍMITE: ELIMINAR EL PRIMER ELEMENTO**

Finalmente, de no presentarse ninguna de las situaciones anteriores, se considera el caso general. Se recorre la lista y se obtiene la referencia al nodo anterior al nodo a eliminar. Sin embargo, existe la posibilidad de que la posición proporcionada por el usuario se encuentre fuera del rango de las posiciones de la lista. Por lo tanto, se considera este caso límite y se indica al usuario la imposibilidad de realizar la eliminación.

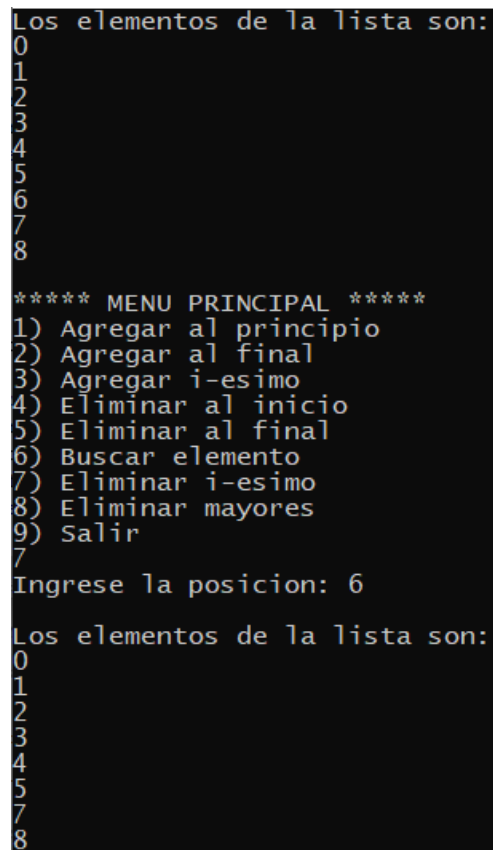
De ser exitosa la obtención del nodo mencionado, se crea una variable auxiliar para almacenar al nodo *a eliminar*, es decir, el nodo sucesor de **temp**. Se “*liga*” el nodo **temp** con el nodo siguiente al nodo *aEliminar*, y finalmente se libera la memoria de éste.

```
for(i=0; i<posicion-1; i++)
{
    // Corner case: la posicion solicitada se encuentra fuera del rango
    if(!temp->next)
    {
        printf("La posicion proporcionada supera el tamaño de la lista. \n");
        return;
    }
    temp=temp->next;
}

Nodo* aEliminar = temp->next;
temp->next = aEliminar->next;
free(aEliminar);
```

#### RECORRIDO DE LA LISTA, CONSIDERACIÓN DE CASO LÍMITE Y ELIMINACIÓN DEL NODO.

A continuación, se muestra una captura de pantalla de la salida del programa realizado para comprobar el comportamiento correcto de la función:



```
Los elementos de la lista son:
0
1
2
3
4
5
6
7
8

***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Agregar i-esimo
4) Eliminar al inicio
5) Eliminar al final
6) Buscar elemento
7) Eliminar i-esimo
8) Eliminar mayores
9) Salir
7
Ingrese la posicion: 6

Los elementos de la lista son:
0
1
2
3
4
5
7
8
```

SALIDA DEL PROGRAMA

d) Elabora la función “eliminar mayores” la cual eliminará todos los elementos mayores a un valor dado por el usuario, en caso de no eliminar nada el programa lo deberá notificar, en caso contrario deberá indicar cuantos y cuales elementos fueron eliminados. Se deberá suponer que la lista donde se van a eliminar elementos ya existe. Comprueba tu función en un programa (también agrégala al menú).

La función se implementó de la siguiente forma:

Primero, se solicita el número referencia o límite al usuario. Se considera el caso límite de una lista vacía: no es posible eliminar ningún elemento, por lo tanto, se le notifica al usuario y la función finaliza su ejecución. De lo contrario, la lista contiene potenciales elementos a ser eliminados.

Se crean las variables auxiliares **eliminados**, que funge como acumulador para el número de elementos eliminados, **trav**: una referencia al nodo que está siendo analizado (se inicializa con *head*), y **prev**, que contiene una referencia al nodo anterior a *trav*. Esta última se inicializa con el valor NULL, ya que no hay ningún nodo anterior a *head*.

También se imprime el texto “Valores eliminados: ”, ya que en la siguiente parte del código se imprimirán los elementos eliminados, en caso de existir.

```
void eliminarMayores(Lista *lista)
{
    // Solicitar limite
    int limite;
    printf("Eliminar numeros mayores a: ");
    scanf("%d", &limite);

    if (!lista->head)
    {
        printf("\n0 elementos eliminados (lista vacia)\n");
        return;
    }

    int eliminados = 0;
    Nodo *prev = NULL;
    Nodo *trav = lista->head;
    printf("\nValores eliminados: \n");
```

#### SOLICITUD DE LÍMITE, CONSIDERACIÓN DE CASO BASE E INICIALIZACIÓN DE VARIABLES

Posteriormente, se recorre la lista utilizando el nodo auxiliar **trav**. En cada iteración, se pueden presentar **dos casos**: el nodo en cuestión se eliminará, o bien, únicamente se recorrerá el nodo **trav** al nodo siguiente.

En el primer caso, se imprime el valor del elemento a eliminar y se incrementa la variable **eliminados**. Después, se consideran dos posibles subcasos: el nodo a eliminar es el



nodo **head**, o bien, el nodo a eliminar es diferente a éste. En ambos casos se actualizan los nodos **prev** y **trav** a sus nuevos valores correspondientes, y se elimina el nodo respectivo.

```
while(trav)
{
    // Eliminar nodo
    if (trav->val > limite)
    {
        printf("%d\n", trav->val);
        eliminados++;

        // Eliminar head
        if (!prev)
        {
            prev = trav;
            trav = trav->next;
            lista->head = trav;
            free(prev);
            prev = NULL;
        }
        // Nodo a eliminar es diferente de head
        else
        {
            prev->next = trav->next;
            free(trav);
            trav = prev->next;
        }
    }
}
```

#### PRIMER CASO: ELIMINACIÓN DEL NODO

En el segundo caso, únicamente se actualizan los nodos **prev** y **trav**, de tal forma que se “recorre” una posición en la lista.

```
// Avanzar un nodo
else
{
    prev = trav;
    trav = trav->next;
}
```

#### SEGUNDO CASO: RECORRER EL NODO TRAV

Finalmente, se imprime el número total de elementos eliminados:

```
printf("\n%d elemento(s) eliminados\n", eliminados);
```

#### SE IMPRIME EL NÚMERO TOTAL DE NODOS ELIMINADOS

A continuación, se muestra una captura de pantalla de la salida del programa realizado para comprobar el comportamiento correcto de la función:

```

Los elementos de la lista son:
0
1
2
3
4
5
6
7
8
9

***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Agregar i-esimo
4) Eliminar al inicio
5) Eliminar al final
6) Buscar elemento
7) Eliminar i-esimo
8) Eliminar mayores
9) Salir
8
Eliminar numeros mayores a: 5
valores eliminados:
6
7
8
9

4 elemento(s) eliminados
Los elementos de la lista son:
0
1
2
3
4
5

```

#### SALIDA DEL PROGRAMA

#### e) Explica cuál fue la mayor dificultad de este ejercicio

En mi opinión, las complicaciones más notables en esta actividad se presentaron en las funciones **agregar enésimo** y **eliminar enésimo**. Lo anterior fue debido a la consideración de los casos límite: fue necesario encontrar una manera de cumplir con el objetivo de las funciones de tal forma que ésta fuera consistente para cualquier caso.

De igual forma, en la función **eliminarMayores** también fue necesario considerar todos los casos posibles. Además, la implementación de la lógica de eliminación de los nodos también fue un aspecto un tanto complicado.

## ACTIVIDAD 2

a) Elabora un programa con las instrucciones que tú quieras para comprobar el funcionamiento de las funciones que desees de la biblioteca *listaCircular.h* (puede ser un menú como la actividad anterior).

La biblioteca contenía los siguientes errores:

El tipo de dato **Nodo** no se encontraba definido en la sentencia **typedef**, por lo que fue necesario corregirlo.

```
typedef struct Nodo{
    int val;
    struct Nodo* next;
}Nodo;
```

DEFINICIÓN DE ESTRUCTURA NODO

Las funciones **borrarUltimo** y **borrarPrimero** de la biblioteca presentaron errores: no cumplían con su objetivo. Fue necesario realizar las siguientes modificaciones:

```
void borrarPrimero(Lista *lista) {
    if (lista->head == NULL) {
        printf("La lista esta vacia");
    }
    else{
        lista->tamano--;
        Nodo *temp = lista->head;
        Nodo *nuevo_head = (lista->tamano == 0) ? NULL : temp->next;
        while(temp->next != lista->head){
            temp=temp->next;
        }
        temp->next = nuevo_head;
        free(lista->head);
        lista->head = nuevo_head;
    }
}
```

FUNCIÓN BORRARPRIMERO

En el caso general (es decir, una lista no vacía), el nodo en cuestión era eliminado, pero el último nodo en la lista no era actualizado, y, además, el decremento en el tamaño de la lista tampoco era realizado. Se realizaron estas correcciones, además de la consideración del caso de una lista de un elemento, donde el nodo **head** debe actualizado a **NULL**.

```

void borrarUltimo(Lista *lista) {
    if (lista->head == NULL) {
        printf("La lista esta vacia");
    }
    else{
        if (lista->tamano == 1){
            borrarPrimero(lista);
            return;
        }
        lista->tamano--;
        Nodo *current = lista->head;
        Nodo *current2 = current->next;

        while (current2->next != lista->head) {
            current = current->next;
            current2 = current2->next;
        }
        current->next = lista->head;
        free(current2);
    }
}

```

#### FUNCIÓN BORRARULTIMO

En este caso, la eliminación no se llevaba a cabo de forma correcta: fue necesario reescribir la función. Inicialmente, se consideraron dos casos: una lista vacía y una lista no vacía. En el primer caso, únicamente se le indica al usuario que la lista se encuentra vacía.

En el caso general (*lista no vacía*), primero se considera el subcaso de una lista de longitud 1. Por simplicidad, se delega esta tarea a la función **borrarPrimero**, que ya cuenta con soporte para este caso. De no presentarse este caso, la función continúa su ejecución. Se realiza el decremento al tamaño de la lista, se recorre la lista hasta obtener una referencia del penúltimo nodo y se realiza la eliminación correspondiente.

A continuación, se muestra la salida del programa realizado para comprobar el correcto comportamiento de la función:

```

Los elementos de la lista son:
1
2

***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Eliminar al inicio
4) Eliminar al final
5) Salir
3

Los elementos de la lista son:
2

***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Eliminar al inicio
4) Eliminar al final
5) Salir
4

LA LISTA ESTA VACIA

```

SALIDA DEL PROGRAMA

b) Se requiere diseñar y elaborar un tipo de dato abstracto para modelar Automóvil, los miembros del TDA, así como las funciones son libres de diseño. (Marca, modelo, placas, color, pasajeros, transmisión, etc.)

La estructura Automóvil se definió en la biblioteca de la siguiente forma:

```

typedef struct
{
    char marca[100];
    char modelo[100];
    char color[100];
    char placas[100];
} Automovil;

```

ESTRUCTURA AUTOMÓVIL

Se definieron las siguientes 4 operaciones para este tipo de dato:

```

Automovil *crearAutomovil();
void eliminarAutomovil(Automovil *carro);
char *obtenerMarca(Automovil *carro);
void obtenerInfo(Automovil *carro);

```

OPERACIONES DE AUTOMÓVIL

Los nombres de las funciones mostradas son auto descriptivos. **CrearAutomóvil** solicita al usuario los datos necesarios para crear una nueva instancia de *Automóvil*, y retorna un puntero a éste. **EliminarAutomóvil** únicamente libera la memoria utilizada por el *Automóvil* proporcionado como argumento. **ObtenerMarca** retorna una cadena: la marca del *Automóvil* en cuestión. Y, finalmente, **obtenerInfo** imprime los datos pertenecientes al *Automóvil* proporcionado como argumento.

**c) Realiza las modificaciones en listaCircular.c para que en lugar de que sea una lista de enteros ahora sea una lista de automóviles, realiza la función de búsqueda de tal forma que puedas buscar un automóvil por su “marca”.**

La función se implementó de la siguiente forma:

Primero, se consideró el caso de una lista vacía: únicamente se imprime el mensaje mostrado en la captura de pantalla y se retorna el valor **-1**, indicativo de una posición inexistente.

De no presentarse el caso anterior, se crean las variables auxiliares **trav**, para recorrer la lista, **pos**, que indica la posición en la que se encuentra el nodo *trav* (con índices basados en **0**), y **marcaNoEsIgual**, que se utilizaría para almacenar un valor indicativo de la comparación entre la marca buscada y la marca del automóvil en el nodo *trav*.

```
int buscarElemento(Lista* lista, char *marca)
{
    if (!lista->head)
    {
        printf("No se encontro el elemento (lista vacia) \n");
        return -1;
    }

    Nodo *trav = lista->head;
    int pos = 0;
    int marcaNoEsIgual;
```

#### **FUNCIÓN BUSCAR ELEMENTO. CASO LÍMITE Y CREACIÓN DE VARIABLES AUXILIARES**

Posteriormente, se recorre la lista en búsqueda del elemento de interés. En cada iteración, se compara la marca del nodo *trav* con la marca proporcionada por el usuario, utilizando la función **strcmp**. En caso de ser iguales, ésta retorna el valor **0**. De presentarse esta situación, se imprime la posición del nodo y la información del automóvil y la función finaliza su ejecución.

En caso de no encontrar ninguna coincidencia, el ciclo finaliza su ejecución al llegar al final de la lista.

```
while(1)
{
    marcaNoEsIgual = strcmp((trav->carro)->marca, marca);

    if (!marcaNoEsIgual)
    {
        printf("\nElemento encontrado en la posicion %d (r
        obtenerInfo(trav->carro);
        free(marca);
        return pos;
    }

    if (trav->next == lista->head) break;

    pos++;
    trav = trav->next;
```

#### RECORRIDO DE LA LISTA. BÚSQUEDA.

Finalmente, si el ciclo finaliza su ejecución, es evidente que no se encontró el elemento solicitado. Por lo tanto, se imprime el mensaje anterior y se retorna el valor -1.

```
free(marca);
printf("No se encontro el elemento\n");
return -1;
```

#### BÚSQUEDA FALLIDA

A continuación, se muestra la salida del programa realizado para comprobar el correcto comportamiento de la función:

<pre>4) Eliminar al final 5) Buscar automovil por marca 6) Salir 5 Marca del automovil: Honda Elemento encontrado en la posicion 2 (posicion con indices basados en cero) Marca: Honda Modelo: sdc Color: fsda Placas: fg</pre>	<pre>AUTOS: AUTO 1 Marca: Nissan Modelo: fw Color: sadfg Placas: fds AUTO 2 Marca: Mazda Modelo: fas Color: verde Placas: daf123 AUTO 3 Marca: Honda Modelo: sdc Color: fsda Placas: fg AUTO 4 Marca: Mitsubishi Modelo: fdsa Color: gg Placas: s</pre>
---	---

#### SALIDA DEL PROGRAMA

d) **Agrega en la biblioteca la función “recorrer lista” de tal forma que el usuario pueda recorrer la lista y elegir entre avanzar, seleccionar el elemento actual para ver sus detalles o salir de la lista circular.**

La función se implementó de la siguiente forma:

Primero, se consideró el caso de una lista vacía, donde únicamente se imprime un mensaje indicando la imposibilidad de recorrerla. De no presentarse, se crean las variables auxiliares **trav**, un nodo auxiliar para recorrer la lista, y **flag**, que fungirá como una bandera para determinar el siguiente paso en el programa. La primera se inicializa con el valor del nodo **head**, y la segunda con el valor **1**, que, al momento de ser evaluado como una expresión booleana es equivalente al valor **true**.

```
void recorrerLista(Lista *lista)
{
    if (!lista->head) {
        printf("No se puede recorrer la lista (vacía)\n");
        return;
    }

    Nodo *trav = lista->head;
    int flag = 1;
    while (flag)
```

**FUNCIÓN RECORRERLISTA. CASO DE LISTA VACÍA Y VARIABLES AUXILIARES.**

Posteriormente, se recorre la lista en el ciclo principal. En cada iteración, se actualiza la variable **flag** con el valor de retorno de la función **solicitarOpcion**. Esta última imprime el menú con 3 opciones: mostrar la información del nodo actual, avanzar al nodo siguiente, y salir del recorrido. Retorna 3 valores posibles: **0**, que indica la salida del recorrido, **1**, que indica la opción de obtener la información del nodo, o **3**, que indica la opción de avanzar al siguiente nodo.

Una vez actualizada la variable **flag**, se ejecuta la opción elegida: si el valor es **1**, se ejecuta un ciclo donde se despliega la información del nodo y se vuelve a solicitar la opción a elegir. En caso del valor **2**, se avanza al siguiente nodo y se ejecuta otra iteración del ciclo principal, ya que el número **2** se evalúa como el valor booleano **true**. En el último caso (el valor de **flag** es cero), el ciclo principal finaliza su ejecución, y, por ende, la función también.



```

while (flag)
{
    flag = solicitarOpcion();
    while(flag == 1)
    {
        printf("\nINFORMACION DEL AUTO ACTUAL:\n");
        obtenerInfo(trav->carro);
        flag = solicitarOpcion();
    }

    if (flag == 2)
    {
        printf("\nAVANZANDO AL SIGUIENTE AUTO\n");
        trav = trav->next;
    }
}

```

**CICLO PRINCIPAL, MANEJO DE LAS OPCIONES.**

A continuación, se muestra la salida del programa realizado para comprobar el comportamiento correcto de la función:

AUTOS:	Elija una opcion:
AUTO 1	1) Ver detalles del nodo actual
Marca: Nissan	2) Avanzar al siguiente elemento
Modelo: 21	3) Salir
Color: wfgd	1
Placas: vgb	INFORMACION DEL AUTO ACTUAL:
	Marca: Nissan
	Modelo: 21
	Color: wfgd
	Placas: vgb
AUTO 2	Elija una opcion:
Marca: Mazda	1) Ver detalles del nodo actual
Modelo: fdsa	2) Avanzar al siguiente elemento
Color: gf	3) Salir
Placas: h	2
	AVANZANDO AL SIGUIENTE AUTO
AUTO 3	Elija una opcion:
Marca: Honda	1) Ver detalles del nodo actual
Modelo: edf	2) Avanzar al siguiente elemento
Color: gh	3) Salir
Placas: s	1
AUTO 4	INFORMACION DEL AUTO ACTUAL:
Marca: Suzuki	Marca: Mazda
Modelo: vf	Modelo: fdsa
Color: g	Color: gf
Placas: d	Placas: h

**SALIDA DEL PROGRAMA**

## ACTIVIDAD 3

a) Explica por qué no aparece el primer dato que se agrega a la lista y qué corrección se debe hacer para que aparezca.

En la función **print\_list** se ejecuta un ciclo para recorrer la lista. Sin embargo, el primer nodo que se imprime es el siguiente al nodo **head**, ya que en cada iteración se imprime el nodo siguiente al nodo **current**:

```
Nodo *current = lista.head;
while (current != NULL){
    printf("%d\n", current->next->val) ;
    current = current->next;
}
```

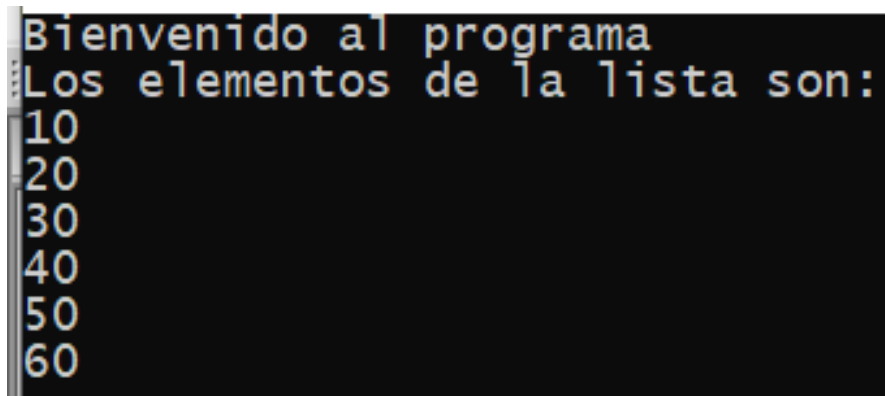
### ERROR EN FUNCIÓN PRINT\_LIST

En cambio, en cada iteración debe imprimirse primero el dato presente en el nodo **current**, y después avanzar al siguiente, hasta llegar al final de la lista:

```
while (current != NULL){
    printf("%d\n", current->val) ;
    current = current->next;
}
```

### CORRECCIÓN

A continuación, se muestra la salida del programa después de realizar la corrección:



```
Bienvenido al programa
Los elementos de la lista son:
10
20
30
40
50
60
```

### SALIDA DEL PROGRAMA

**b) Explica la diferencia entre las funciones *printList* y *printList2*.**

La función **print\_list** imprime los elementos de la lista en el orden convencional, es decir, comenzando por el nodo head.

En cambio, la función **print\_list2** imprime los elementos de la lista de forma inversa. Para lograrlo, primero se recorre la lista hasta encontrar el último nodo. Partiendo de éste, se recorre la lista en orden inverso, imprimiendo la información de cada nodo.

```

Bienvenido al programa
Los elementos de la lista son:
10
20
30
40
50
60

Los elementos de la lista son:
60
50
40
30
20
10

```

IMPRESIÓN DE LA LISTA EN ORDEN NORMAL E INVERSO

**c) Explica para qué sirve la función primer nodo.**

Su función es crear un nodo con el argumento **val** proporcionado, e insertarlo a la lista proporcionada. Se asume que la lista se encuentra **vacía**.

**d) ¿Qué se le tendría que agregar a la función agregar i-ésimo para agregar una validación que no permita agregar una posición que no existe en la lista (Esto sin agregar el miembro “tamaño” a la estructura lista)?**

Primero, se tendría que agregar una condicional para comprobar que la posición dada no fuera un número negativo. En caso de serlo, se debería informar al usuario de ello.

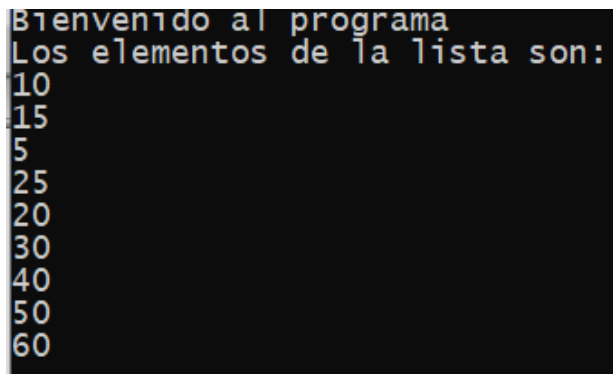
El segundo caso se presentaría cuando la posición proporcionada sobrepasa el tamaño de la lista. Esta situación sería captada dentro del ciclo donde se recorre la lista, cuyo objetivo es “recorrer” el nodo **temp** al nodo anterior a la posición de interés. Por lo tanto, si se llegara al final de la lista antes de finalizar la ejecución del ciclo, la posición solicitada

sería inválida. En otras palabras, esta situación ocurre cuando el nodo **temp** es igual al valor **NULL**, lo cual indica que se ha llegado al final de la lista.

Es importante mencionar que sí es posible insertar un nodo al final de la lista, es decir, cuando la posición es igual al tamaño de ésta, ya que, después de ejecutar el ciclo, el nodo **temp** se “recorre” al último nodo y posteriormente se inserta el nuevo ejemplar.

#### e) Escribe el código para resolver el problema anterior.

Fue necesario modificar la función, ya que su funcionamiento no era el esperado:



```
Bienvenido al programa
Los elementos de la lista son:
10
15
5
25
20
30
40
50
60
```

**ERROR: EL NÚMERO 15 DEBE IR DESPUÉS DEL NÚMERO 5**

Se consideró el caso límite de una inserción en la posición 0 antes de realizar el recorrido de la lista:

```
// Corner case: insertar al principio (posicion 0)
if (posicion == 0)
{
    lista->head = nuevoNodo;
    nuevoNodo->prev = NULL;
    nuevoNodo->next = temp;
    if (temp)
    {
        temp->prev = nuevoNodo;
    }
    return;
}
```

**INSERCIÓN EN LA POSICIÓN 0**

Además, se modificó el valor del contador inicial en el ciclo, y se consideró el caso de la inserción al final de la lista, al momento de insertar el nuevo nodo:

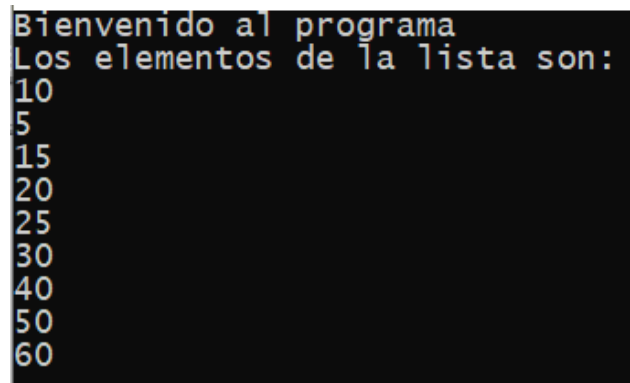
```

for(contador=0;contador<posicion-1;contador++)
    nuevoNodo->next = temp->next;
nuevoNodo->prev = temp;
if (temp->next)
{
    (temp->next)->prev = nuevoNodo;
}
temp->next = nuevoNodo;

```

#### MODIFICACIONES REALIZADAS

Después de estas modificaciones, la salida del programa fue la esperada (*la convención utilizada para las posiciones es de índices basados en cero*):



```

Bienvenido al programa
Los elementos de la lista son:
10
5
15
20
25
30
40
50
60

```

#### SALIDA DEL PROGRAMA

Ahora bien, las modificaciones realizadas para manejar el error de una posición inválida proporcionada fueron las siguientes:

Al inicio de la función se añadió la siguiente condicional, cuyo objetivo es notificar el error en caso de una posición negativa proporcionada. En caso de presentarse esta condición, la función finaliza su ejecución.

```

if (posicion < 0)
{
    printf("La posicion proporcionada es invalida (debe ser un entero positivo)\n");
    return;
}

```

#### CASO DE POSICIÓN NEGATIVA PROPORCIONADA

La otra modificación se realizó dentro del ciclo donde se recorre la lista: si el valor del nodo **temp** es **NULL** en alguna iteración, significa que se ha llegado al final de la lista antes de llegar al nodo deseado, por lo tanto, únicamente se imprime un mensaje con la imposibilidad de realizar la inserción y la función finaliza su ejecución.

```

if(temp==NULL)
{
    printf("La posicion sobrepasa los limites de la lista.\n");
    return;
}

```

#### CASO DE POSICIÓN INVÁLIDA: SOBREPASA EL TAMAÑO DE LA LISTA

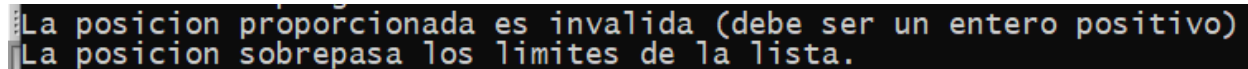
A continuación, se muestra la salida de un programa realizado para comprobar el correcto comportamiento de la función anterior:

```

Lista miLista =crearLista();
addFinallista(&miLista,10);
addIesimoLista(&miLista, 2, -1);
addFinallista(&miLista,20);
addIesimoLista(&miLista, 2, 5);

```

SE INTENTA INGRESAR UN ELEMENTO A LA POSICIÓN -1, Y OTRO ELEMENTO A LA POSICIÓN 5 CUANDO LA LISTA CONTIENE ÚNICAMENTE 2 ELEMENTOS.



```

La posicion proporcionada es invalida (debe ser un entero positivo)
La posicion sobrepasa los limites de la lista.

```

LOS MENSAJES DE ERROR MOSTRADOS EN PANTALLA COINCIDEN CON LAS SITUACIONES MENCIONADAS EN LA IMAGEN ANTERIOR

## ACTIVIDAD 4

a) Completa la función *borrarPrimero* de la biblioteca y elabora un programa para probar esta y las otras funciones de esta lista.

La función se implementó de la siguiente manera:

Se consideraron los dos casos principales: una lista vacía o una lista no vacía. En el primer caso, únicamente se muestra un mensaje indicando la condición mencionada. En el segundo, sí se realiza la eliminación pertinente.

Para realizar la eliminación, inicialmente se considera el subcaso de una lista de un sólo nodo, realizando las asignaciones correspondientes y finalizando la ejecución de la función. De no presentarse el subcaso anterior, de igual forma se realizan las operaciones pertinentes para realizar la eliminación, finalizando así la ejecución de la función. Cabe

mencionar que en ambos subcasos se realiza un decremento al miembro **tamaño**, de la lista en cuestión.

```
void borrarPrimero(Lista *lista) {
    if (lista->head == NULL) {
        printf("La lista esta vacia");
    }
    else{
        Nodo *temp = lista->head;

        if (lista->tamano == 1)
        {
            lista->tamano--;
            lista->head = NULL;
            free(temp);
            return;
        }

        Nodo *tail = temp->prev;
        Nodo *newHead = temp->next;

        tail->next = newHead;
        newHead->prev = tail;
        free(temp);
        lista->head = newHead;
        lista->tamano--;
    }
}
```

#### **FUNCIÓN BORRARPRIMERO**

Para realizar la función **borrarUltimo** se utilizó exactamente la misma lógica, realizando únicamente las modificaciones correspondientes en la asignación de los nodos.

A continuación, se muestra la salida del programa creado para comprobar el correcto comportamiento de las funciones de la biblioteca:

```
***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Eliminar al inicio
4) Eliminar al final
5) Recorrer lista
6) Salir
1
Introduzca el elemento: 0
Los elementos de la lista son:
0
***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Eliminar al inicio
4) Eliminar al final
5) Recorrer lista
6) Salir
2
Introduzca el elemento: 1
Los elementos de la lista son:
0
1
```

#### **INSERCIÓN DE DOS ELEMENTOS EN LA LISTA**

```

***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Eliminar al inicio
4) Eliminar al final
5) Recorrer lista
6) Salir
3

Los elementos de la lista son:
1

***** MENU PRINCIPAL *****
1) Agregar al principio
2) Agregar al final
3) Eliminar al inicio
4) Eliminar al final
5) Recorrer lista
6) Salir
4

LA LISTA ESTA VACIA

```

#### ELIMINACIÓN DE LOS ELEMENTOS INSERTADOS ANTERIORMENTE

b) Modifica la función *recorrerLista* de manera tal que el usuario pueda elegir si se recorre hacia “adelante” o se recorre hacia “atrás”. Escribe el código realizado.

Para implementar la funcionalidad, se modificó únicamente la parte del ciclo principal de la función, donde se recorre la lista. Se añadió la opción en el menú impreso, con el mensaje “Para mostrar el elemento anterior presione 0”. Finalmente, se agregó una condicional para modificar el nodo **temp** según la opción elegida por el usuario, es decir, se avanzaba o retrocedía al siguiente nodo dependiendo de lo anterior.

```

do{
    printf("El elemento actual es %d \n", temp->val);
    printf("Mostrar anterior presione 0\n");
    printf("Mostrar Siguiente Presione 1 \n");
    printf("Salir Presione 2 \n");
    scanf("%d", &var);
    if (var == 0)
    {
        temp = temp->prev;
    }
    else if(var == 1)
    {
        temp = temp->next;
    }
}while (var!=2);

```

#### MODIFICACIÓN DE LA FUNCIÓN RECORRERLISTA



## CONCLUSIONES

Gracias a la realización de la práctica, logré reafirmar ampliamente mis conocimientos sobre los cuatro tipos de listas, analizando sus definiciones, características y comportamiento en distintos casos.

Al realizar las implementaciones o correcciones de las funciones propias de las estructuras, logré obtener un conocimiento más profundo de su funcionamiento interno. Cabe mencionar que dichas tareas no fueron sencillas de realizar, debido a la inherente complejidad de los diferentes tipos de listas. En mi opinión, uno de los aspectos más complicados fue la consideración de todos los casos posibles para cada estructura, además de la identificación de los errores presentes en las funciones.

En cada ejercicio se implementaron nuevas funcionalidades a las listas, además de las modificaciones realizadas a las funciones ya existentes en las bibliotecas correspondientes. Además, en las actividades 1, 2 y 4 se creó un programa interactivo con un menú cuyas opciones permitían al usuario utilizar las funciones propias de cada estructura, y las funciones realizadas en cada actividad. Lo anterior facilitó en gran medida la identificación de errores en los programas, además de proporcionar una interfaz clara para la manipulación de la estructura correspondiente.

Si bien los cuatro tipos de listas cuentan con muchas similitudes, es necesario identificar sus implicaciones para poder determinar cuál es la más apta para realizar la tarea deseada. Cabe mencionar que es posible implementar cualquier operación de lista en cualquier tipo de lista, sin embargo, es probable que la eficiencia de algunas operaciones no sea óptima utilizando la estructura incorrecta. Por ejemplo, en las listas simples no es posible “retroceder”, a diferencia de las listas dobles. Es posible implementar la operación anterior en una lista simple, sin embargo, su eficiencia sería mala. En este caso, es evidente que la utilización de una lista doble es mejor. Por otro lado, se puede decir que las listas dobles utilizan más memoria que las listas simples. El punto anterior es un claro ejemplo de la necesidad de evaluar el problema de interés, para así utilizar una estructura que sea óptima para su resolución.

Las listas son una de las estructuras de datos más importantes y básicas en la Computación; es evidente que tienen una numerosa cantidad de aplicaciones, he ahí la gran importancia de comprender su funcionamiento, implementación y características.

Los ejercicios realizados en la práctica fueron adecuados para comprender de forma profunda el funcionamiento e implementación de las estructuras. Sin embargo, una posible mejora podría ser realizar la implementación de las estructuras desde cero, o bien, únicamente proporcionar la biblioteca con funciones incompletas, que consideren solamente el caso general para cada operación.