
**Universidad Cenfotec**

**Autor(es):**

-Isabel Galeano Hernández, Cédula: 1-1888-0968

-José García Quirós, Cédula: 1-1782-0076

-Orlando Trejos Montano, Cédula: 7-0281-0993

-Daniel Zúñiga Rojas, Cédula: 1-1811-0097

**Tema:**
Investigación Pilas y Colas

**Curso:**
BISOFT-11  Estructuras de datos

**Profesor:**
Christian Sibaja Fernández

**Fecha de entrega:**
18/07/2021

**Cuatrimestre:**
2021-2

# Índice

# Lista circular

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 almacenar 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.

## I. Crear una lista circular

Diseñar un algoritmo que permita crear una lista circular sencilla con N número de nodos. Para resolver este problema es necesario la utilización de un ciclo que estará generando cada uno de los nodos que formarán 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.

## II. Listar lista circular

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

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.

## III. Insertar al inicio de la lista circular

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

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.

## IV. Insertar al fInal de la lista circular

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

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.

## V. Insertar en cualquier posición de la lista circular

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

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: si se desea insertar el nuevo nodo en una posición en el medio de dos nodos ya existentes, cuando se quiere insertar un nodo al inicio de la lista y cuando se quiera insertar un nodo al final de la lista.

## VI. Borrar un nodo al final de la lista circular

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

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.

## VII. Borrar un nodo al inicio de la lista circular

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

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.

## VIII. Borrar nodo en cualquier posición de la lista circular

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

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.

## IX. Ordenar nodo en lista circular el método burbuja

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

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.

## X. Buscar un elemento en la lista circular

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

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.

## XI. Borrar la lista circular

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

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.

# Lista doblemente enlazada

Una lista doblemente enlazada se caracteriza debido a que cada nodo tiene dos enlaces, uno al nodo siguiente, y otro al anterior.

Las listas doblemente enlazadas no necesitan un nodo especial para acceder a ellas, pueden recorrerse en ambos sentidos a partir de cualquier nodo, esto es porque a partir de cualquier nodo, siempre es posible alcanzar cualquier nodo de la lista, hasta que se llega a uno de los extremos.

Como la lista doblemente enlazada contiene dos punteros, es decir, anterior y siguiente, podemos recorrerla en las direcciones hacia adelante y hacia atrás. Esta es la principal ventaja de la lista doblemente enlazada sobre la lista enlazada individualmente

Al igual que las listas simples, las listas doblemente enlazadas también tienen métodos parecidos, algunos de ellos son:

## I. Añadir o insertar elementos.

Para insertar un elemento en la primera posición partimos de una lista no vacía. Para simplificar, consideraremos que lista apunte al primer elemento de la lista doblemente enlazada. El proceso es el siguiente:

1. Si lista está vacía hacemos que Lista apunte a nodo. Y nodo->anterior y nodo->siguiente a NULL.
2. Si lista no está vacía, hacemos que nodo->siguiente apunte a Lista->siguiente.
3. Después que Lista->siguiente apunte a nodo.
4. Hacemos que nodo->anterior apunte a Lista.
5. Si nodo->siguiente no es NULL, entonces hacemos que nodo->siguiente->anterior apunte a nodo.

## II. Eliminar elementos.

El nodo a borrar esté apuntado por Lista o no. Si lo está, simplemente hacemos que Lista sea Lista->anterior, si no es NULL o Lista->siguiente en caso contrario.

1. Si nodo apunta a Lista,
    - Si Lista->anterior no es NULL hacemos que Lista apunte a Lista->anterior.

- Si Lista->siguiente no es NULL hacemos que Lista apunte a Lista->siguiente.
- Si ambos son NULL, hacemos que Lista sea NULL.

2. Si nodo->anterior no es NULL, hacemos que nodo->anterior->siguiente apunte a nodo->siguiente.
3. Si nodo->siguiente no es NULL, hacemos que nodo->siguiente->anterior apunte a nodo->anterior.
4. Borramos el nodo apuntado por nodo.

## III. Ejemplo de una lista doblemente enlazada.

**Nodo.H**

```
#pragma once
#ifndef NODO_H
#define NODO_H
class Nodo
{

public:
	Nodo();
	Nodo(int dato);
	void setDato(int dato);
	int getDato();
	void setSiguiente(Nodo* siguiente);
	Nodo* getSiguiente();
	void setAnterior(Nodo* anterior);
	Nodo* getAnterior();
private:
	int dato;
	Nodo* siguiente;
	Nodo* anterior;

};

#endif // NODO_H
```

**Lista.H**

```
#pragma once
#ifndef LISTA_H
#define LISTA_H
#include "Nodo.h"
#include <string>
class Lista
{
public:
	Lista();
	void setCabeza(Nodo* cabeza);
	Nodo* getCabeza();
	bool esVacia();
	void insertar(int x);
	void mostrar();
	void eliminar(int x);
	int size = 0;
private:
	Nodo* cabeza;

};

#endif // LISTA_H
```

**Lista.cpp**

```
#include "Lista.h"
#include <iostream>
using namespace std;
Lista::Lista() {
	cabeza = nullptr;
}

bool Lista::esVacia() {
	if (cabeza == nullptr)
	{
		return true;
	}

	return false;
}
```

```
void Lista::insertar(int num) {

	Nodo* nuevoNodo = new Nodo(num);

	if (cabeza == nullptr)
	{
		cabeza = nuevoNodo;
	}
	else
	{
		nuevoNodo->setSiguiente(cabeza);
		cabeza->setAnterior(nuevoNodo);
		cabeza = nuevoNodo;
		size++;
	}
}

void Lista::mostrar() {
	Nodo* aux = this->cabeza;
	if (cabeza == nullptr)
	{
		cout << "La lista esta vacia" << endl;
	}
	else
	{
		while (aux->getSiguiente() != nullptr)
		{
			cout << aux->getDato() << endl;
			aux = aux->getSiguiente();
		}
	}
}

void Lista::eliminar(int pos) {
	if (pos <= size)
	{
		Nodo* aux;
		if (pos == 1)
		{
			aux = cabeza;
			cabeza = cabeza->getSiguiente();
			if (cabeza != nullptr)
			{
				cabeza->setAnterior(nullptr);
			}
		}
		else
		{
			Nodo* aux2;
```

```
                        aux2 = cabeza;
                        for (int i = 1; i <= pos - 2; i++)
                        {
                                aux2 = aux2->getSiguiente();
                        }
                        Nodo* aux3 = aux2->getSiguiente();
                        aux = aux3;
                        aux2->setSiguiente(aux3->getSiguiente());
                        Nodo* siguiente = aux2->getSiguiente();
                        if (siguiente != NULL)
                        {
                                siguiente->setAnterior(aux2);
                        }


                }

                delete aux;
        }
}
```

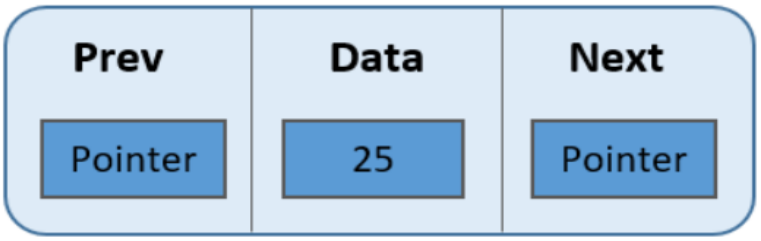
**Nodo.cpp**

```
#include "Nodo.h"

Nodo::Nodo() {

}

Nodo::Nodo(int _dato) {
        dato = _dato;
}

void Nodo::setDato(int _dato) {
        dato = _dato;
}

int Nodo::getDato() {
        return dato;
}

void Nodo::setSiguiente(Nodo* _siguiente) {
        siguiente = _siguiente;
}

void Nodo::setAnterior(Nodo* _anterior) {
        this->anterior = _anterior;
```

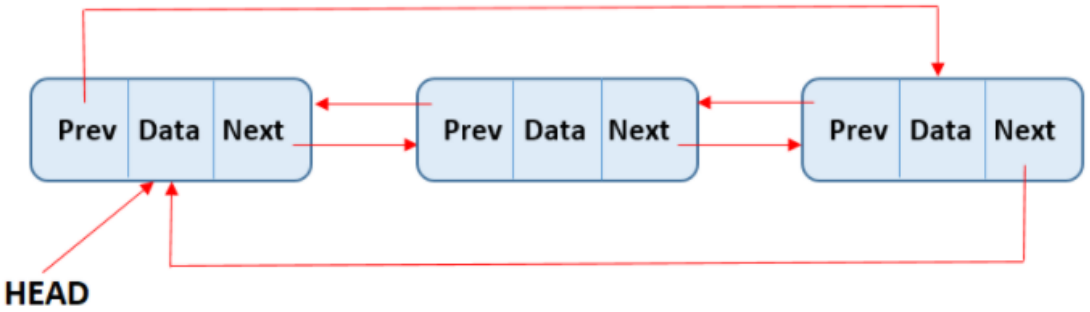
```
}

Nodo* Nodo::getAnterior() {
	return anterior;
}


Nodo* Nodo::getSiguiente() {
	return siguiente;
}
```

## Lista circular doblemente enlazada

Una lista circular doblemente enlazada es una estructura de datos lineal, en la que los elementos se almacenan en forma de nodo. Cada nodo contiene tres subelementos. Una parte de datos que almacena el valor del elemento, la parte anterior que almacena el puntero al nodo anterior y la parte siguiente que almacena el puntero al siguiente nodo como se muestra en la siguiente imagen:



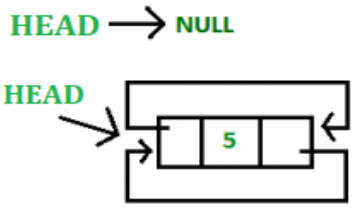

El primer nodo también conocido como cabeza (head) se usa siempre como referencia para recorrer la lista. El último elemento contiene el enlace al primer elemento como el siguiente y el primer elemento contiene el enlace del último elemento como el anterior. Una circular doblemente enlazada se puede visualizar como una cadena de nodos, donde cada nodo apunta al nodo anterior y al siguiente.



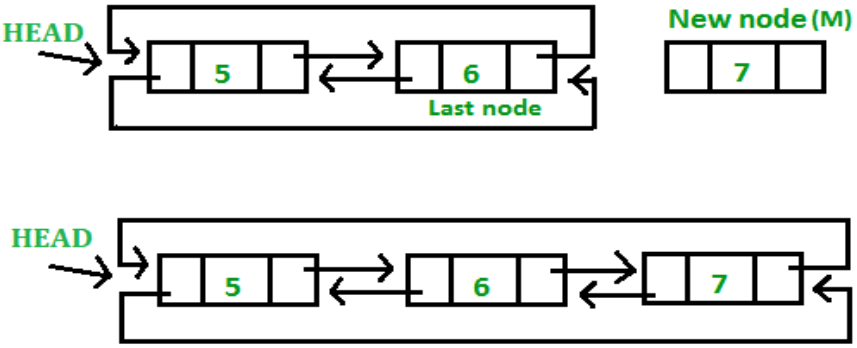

### Inserción en lista circular doblemente enlazada

### Inserción al final de la lista o en una lista vacía

Lista vacía (head = NULO): un nodo (N) se inserta con datos = 5, por lo que el puntero anterior de N apunta a N y el siguiente puntero de N también apunta a N. Pero ahora el puntero de inicio apunta al primer nodo de la lista .

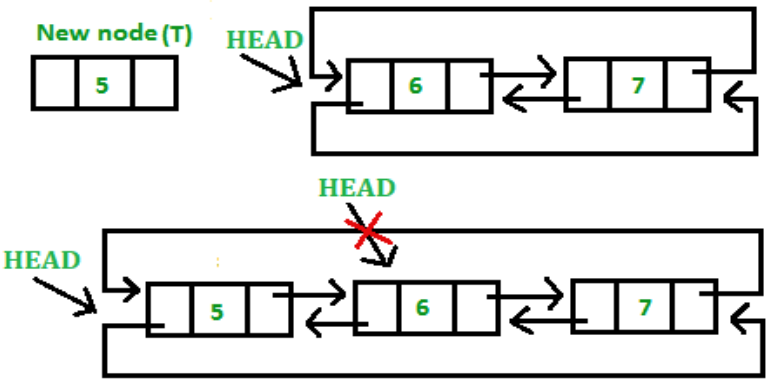



La lista contiene inicialmente algunos nodos, los puntos de inicio al primer nodo de la Lista: un nodo (M) se inserta con datos = 7, por lo que el puntero anterior de M apunta al último nodo, el siguiente puntero de M apunta al primer nodo y al siguiente del último nodo el puntero apunta a este nodo M y el puntero anterior del primer nodo apunta a este nodo M.



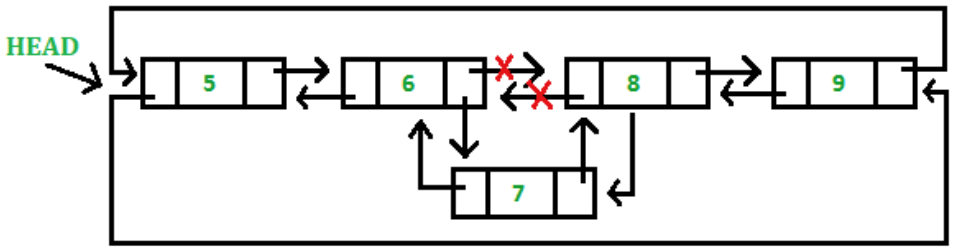

**Inserción al principio de la lista**

Para insertar un nodo al principio de la lista, cree un nodo (Diga T) con datos = 5, T el puntero siguiente apunta al primer nodo de la lista, T el puntero anterior apunta al último nodo de lista, el siguiente puntero del último nodo apunta a este nodo T, el puntero anterior del primer nodo también apunta a este nodo T y, por último, desplazar el puntero 'Inicio' a este nodo T.



**Inserción entre los nodos de la lista**

Para insertar un nodo entre la lista, se requieren dos valores de datos, uno después del cual se insertará un nuevo nodo y otro son los datos del nuevo nodo.




**Eliminar en una lista circular doblemente enlazada**

La Lista contiene inicialmente algunos nodos, puntos de inicio en el primer nodo de la Lista.

1. Si la lista no está vacía, definimos dos punteros curr y previo1 e inicializamos el puntero curr apunta al primer nodo de la lista y previo1 = NULL.

2. Recorra la lista usando el puntero curr para encontrar el nodo que se va a eliminar y antes de pasar de curr al siguiente nodo, cada vez que establezca previo1 = curr.

3. Si se encuentra el nodo, compruebe si es el único nodo de la lista. Si es así, establezca start = NULL y libere el nodo que apunta con curr.

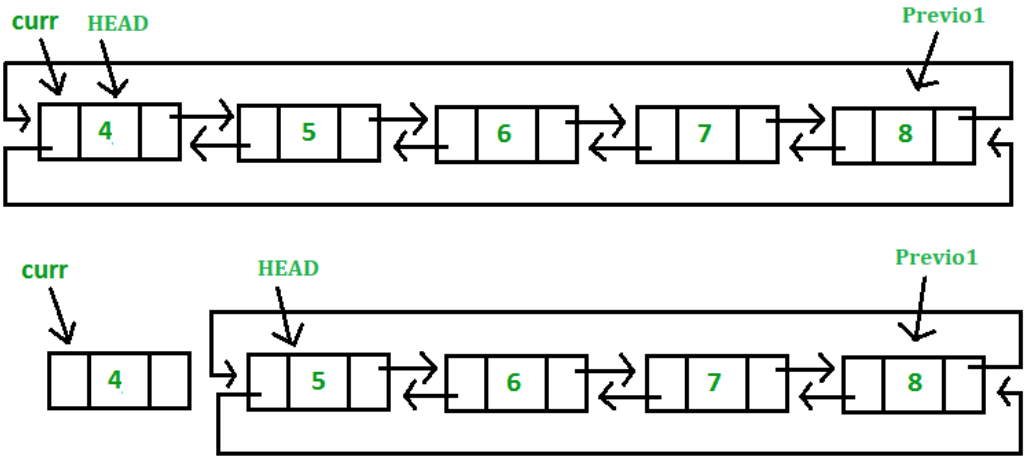
4. Si la lista tiene más de un nodo, verifique si es el primer nodo de la lista. La condición para verificar esto es (curr == inicio). Si es así, mueva previo1 al último nodo (previo1 = inicio -> prev). Después de que previo1 llegue al último nodo, configure start = start -> next y previo1 -> next = start and start -> prev = previo1. Libere el nodo apuntado por curr.

5. Si curr no es el primer nodo, verificamos si es el último nodo de la lista. La condición para verificar esto es (curr -> next == start). En caso afirmativo, establezca previo1 -> siguiente = iniciar y comenzar -> prev = previo1. Libere el nodo apuntando por curr.

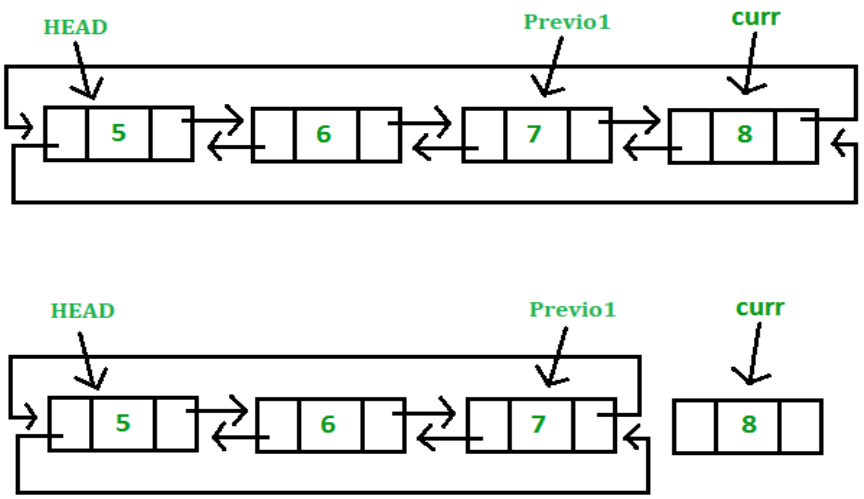
6. Si el nodo que se va a eliminar no es ni el primer nodo ni el último nodo, declare un puntero temp más e inicialice los puntos temp del puntero al

siguiente del puntero curr (temp = curr-> next). Ahora establezca, previo1 -> siguiente = temp y temp -> prev = previo1. Libere el nodo apuntando por curr.

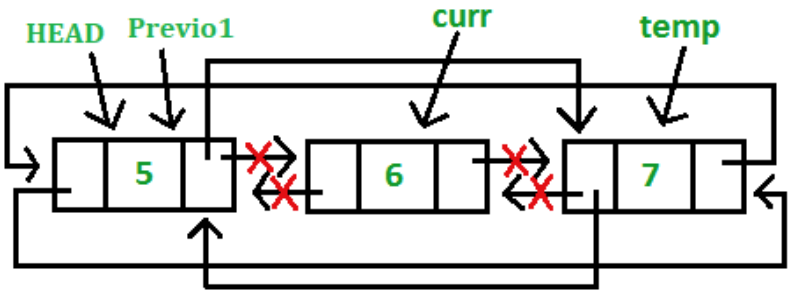
Si la clave dada (4) coincide con el primer nodo de la lista (Paso 4):




Si la clave dada (8) coincide con el último nodo de la lista (Paso 5):




Si la clave dada (6) coincide con el nodo del medio de la lista (Paso 6):



## Bicola

Para entender primero la estructura bicola, es necesario entender las colas sencillas primero. Las bicolas son dos colas por lados contrarios para generar mayor flexibilidad y control de la estructura. Las colas son colecciones ordenadas de elementos de cualquier tipo, ya sea integer, string, boolean, un objeto, etc, y que permite insertar elementos por un lado llamada cola y borrarlos mediante otro extremo llamado cabeza.

“Una cola doble (bicola) es una estructura de datos tipo cola simple en la cual las operaciones ENCOLAR y DESENCOLAR se pueden realizar por ambos extremos de la estructura, es decir, en una cola doble se pueden realizar las operaciones: encolar por head (cabeza), desencolar por head, encolar por tail (cola) y desencolar por tail.

La cola doble es una mejora de una cola simple debido a que es posible realizar operaciones de inserción por ambos extremos de la estructura, permitiendo con esto utilizar el máximo espacio disponible de la estructura. Para poder diseñar un programa que defina el comportamiento de una cola doble se deben considerar 3 casos para las 4 operaciones (Insertar y Eliminar tanto por T como por H): estructura vacía (caso extremo), estructura llena (caso extremo) y estructura con elemento(s) (caso base).” (García y Solano, s.f).

Los casos de las colas definen los comportamientos. La cola doble vacía tiene ambas referencias o extremos (cola y cabeza) apuntando a null. En este caso, se inhabilita desencolar por la falta de elementos pero si se puede encolar por ambos extremos.

La cola doble llena se alcanza el máximo de almacenamiento posible. No es posible encolar elementos por ningún extremo pero si desencolar. La cola doble con elementos es la más común y es cuando la estructura no alcanza su máximo tamaño o almacenamiento. En este caso se pueden desencolar y encolar elementos por ambos extremos.

Las colas dobles tienen dos variantes: bicola con entrada registrada y bicola con salida restringida. Las bicolas con entradas registradas permiten eliminaciones por cualquier extremo (cola o cabeza) pero las inserciones solo se pueden realizar por la cola, osea, el final. La bicola con salida restringida permite insertar elementos por cualquier extremo (cola o cabeza) pero solo permite eliminar por el frente (cola).

Ejemplo de código:

```
struct BICOLA
{
int nodos;
```

```
struct NODO *primero;
struct NODO *ultimo;
};



/* FUNCIONES ****************************************************************/

// Pone los punteros de una bicola a NULL
void inicializarBicola( struct BICOLA **bicola )
{
/*
Precondición:
Se recibe un doble puntero de tipo struct BICOLA a una Bicola sin inicializar..
Poscondición:
Se inicializa la Bicola poniéndola a NULL o 0 según el parámetro a inicializar.
*/

// Se solicita memoria al sistema para la nueva Bicola:
struct BICOLA *temp = (struct BICOLA *) malloc(sizeof(struct BICOLA));

temp->nodos = 0;
temp->primero = NULL;
temp->ultimo = NULL;

(*bicola) = temp;

};



// Inserta nodos por la izquierda:
void insertIzqBicola( struct BICOLA **bicola, int dato )
{
/*
Precondición:
Se recibe un doble puntero de tipo struct BICOLA a una Bicola y un parámetro de tipo entero con el dato a introducir.
Poscondición:
Se inserta al principio un nuevo Nodo en la Bicola.
*/

// Se solicita memoria al sistema para el nuevo Nodo:
struct NODO *temp = (struct NODO *) malloc(sizeof(struct NODO));



// Si la Bicola no tiene Nodos se inserta sin mas
```

```
if( (*bicola)->primero == NULL )
{
temp->elemento.num = dato; // Se guarda el nuevo dato en el nuevo Nodo.
temp->p_anterior = NULL; // Como es el primer Nodo de la Bicola, tanto el puntero anterior como el siguiente apuntan a NULL.
temp->p_siguiente = NULL;

(*bicola)->primero = temp; // Como es el primer Nodo de la Bicola, se hace que primero y ultimo apunten ambos al mismo Nodo.
(*bicola)->ultimo = temp;
}
else
{
temp->elemento.num = dato; // Se guarda el dato nuevo en el nuevo Nodo.
temp->p_anterior = NULL; // Como el nuevo Nodo va a ser el primero de la Bicola no apuntara a ningún Nodo anterior.
temp->p_siguiente = (*bicola)->primero; // El nuevo Nodo apuntara al siguiente Nodo (donde ahora apunta "primero".

(*bicola)->primero->p_anterior = temp; // El puntero anterior del primer Nodo se hace que apunte al nuevo Nodo.
(*bicola)->primero = temp; // El puntero del primero se hace que apunte al nuevo Nodo que ahora es el primero.
};


(*bicola)->nodos += 1; // Se suma 1 a la variable que guarda el numero de Nodos de la Bicola.

};



// Inserta nodos por la derecha:
void insertDerBicola( struct BICOLA **bicola, int dato )
{
/*
Precondición:
Se recibe un doble puntero de tipo struct BICOLA a una Bicola y un parámetro de tipo entero con el dato a introducir.
Poscondición:
Se inserta al final un nuevo Nodo en la Bicola.
*/


struct NODO *temp = (struct NODO *) malloc(sizeof(struct NODO));
```

```
// Si la bicola esta vacía se introduce sin mas:
if( (*bicola)->primero == NULL )
{
temp->elemento.num = dato;
temp->p_anterior = NULL;
temp->p_siguiente = NULL;

(*bicola)->primero = temp;
(*bicola)->ultimo = temp;
}
  // Si contiene nodos, se introduce al final de todos ellos:
else
{
temp->elemento.num = dato;
temp->p_anterior = (*bicola)->ultimo;
temp->p_siguiente = NULL;

(*bicola)->ultimo->p_siguiente = temp;
(*bicola)->ultimo = temp;
};

(*bicola)->nodos += 1;

};



// Elimina el primer Nodo:
void eliminaIzqBicola( struct BICOLA **bicola )
{
/*
Precondición:
Se recibe un doble puntero de tipo struct BICOLA a una Bicola.
Poscondición:
Se elimina el primer Nodo de la Bicola.
*/

struct NODO *aBorrar;

// Si la Bicola esta vacía...
if( (*bicola)->primero == NULL )
```

```
{
printf( "No puede eliminar Nodos de una Bicola vacía." );
}
  // Si solo hay un Nodo en la Bicola, al liberar la RAM del Nodo se inicializa la Bicola a los valores por defecto:
else if( (*bicola)->nodos == 1 )
{
free(*bicola);
inicializarBicola( bicola );
}
  // En caso contrario se elimina solo el primer Nodo:
else if( (*bicola)->nodos > 1 )
{
aBorrar = (*bicola)->primero;

(*bicola)->primero->p_siguiente->p_anterior = NULL;
(*bicola)->primero = (*bicola)->primero->p_siguiente;
free(aBorrar);

(*bicola)->nodos -= 1;
};

};



// Elimina el ultimo Nodo:
void eliminaDerBicola( struct BICOLA **bicola )
{
/*
Precondición:
Se recibe un doble puntero de tipo struct BICOLA a una Bicola.
Poscondición:
Se elimina el ultimo Nodo de la Bicola.
*/


struct NODO *aBorrar;


// Si la Bicola esta vacia...
if( (*bicola)->primero == NULL )
{
printf( "No puede eliminar Nodos de una Bicola vacía." );
}
```

```
 // Si solo hay un Nodo en la Bicola, al liberar la RAM del Nodo se inicializa la Bicola a los valores por defecto:
else if( (*bicola)->nodos == 1 )
{
free(*bicola);
inicializarBicola( bicola );
}
else if( (*bicola)->nodos > 1 )
{
aBorrar = (*bicola)->ultimo;

(*bicola)->ultimo->p_anterior->p_siguiente = NULL;
(*bicola)->ultimo = (*bicola)->ultimo->p_anterior;
free(aBorrar);

(*bicola)->nodos -= 1;
};

};

// Devuelve un true o un false si la Bicola contiene o no Nodos:
int tieneNodosLaBicola( struct BICOLA **bicola )
{
/*
Precondición:
Se recibe un doble puntero de tipo struct BICOLA a una Bicola.
Poscondición:
Se devuelve un parámetro de tipo entero con valores true 1 o false 0 dependiendo de si la Bicola contiene Nodos o no.
*/


int resp = 0;

if( (*bicola)->nodos != 0 )
resp = 1;

return resp;
};


// Libera la memoria RAM usada por la Bicola:
void borraLaBicola( struct BICOLA **bicola )
{
/*
Precondición:
```

```
Se recibe un doble puntero de tipo struct BICOLA a una Bicola.
Poscondición:
Se libera la RAM usada por los Nodos contenidos en la Bicola.
*/


struct NODO *actual, *siguiente;
actual = (*bicola)->primero;

while( actual != NULL )
{
siguiente = actual->p_siguiente;
free(actual);
actual = siguiente;
};

*bicola = NULL;
};
```

## Referencias


1. C++ Program to Implement Circular Doubly Linked List. (s. f.). Recuperado 18 de julio de 2021, de https://www.tutorialspoint.com/cplusplus-program-to-implement-circular-doubly-linked-list
2. C++ Program to Implement Doubly Linked List. (s. f.). Recuperado 18 de julio de 2021, de https://www.tutorialspoint.com/cplusplus-program-to-implement-doubly-linked-list
3. C. (2020, 6 octubre). Listas doblemente enlazadas. Recuperado 18 de julio de 2021, de https://es.ccm.net/faq/2872-listas-doblemente-enlazadas