ESTRUCTURAS DE DATOS

LIC. REDES Y SERVICIOS DE CÓMPUTO

Clase 2

- Unidad IV

Listas ligadas o enlazadas.

Fundamentos teóricos

Clasificación de listas ligadas

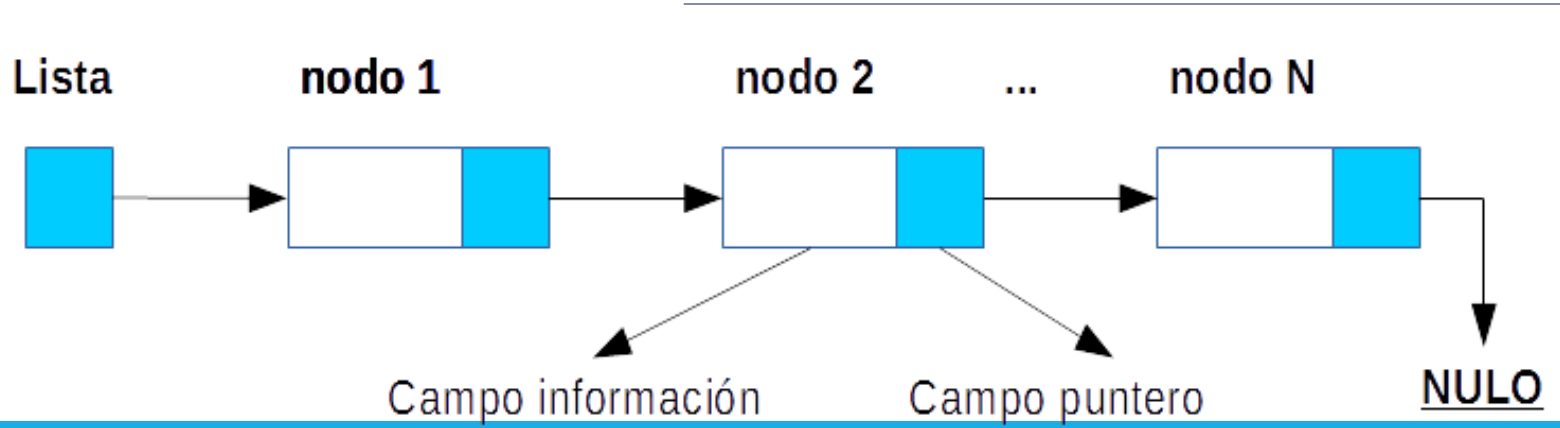
Implantación de listas ligadas

Operaciones básicas de listas ligadas

Operaciones complementarias de listas

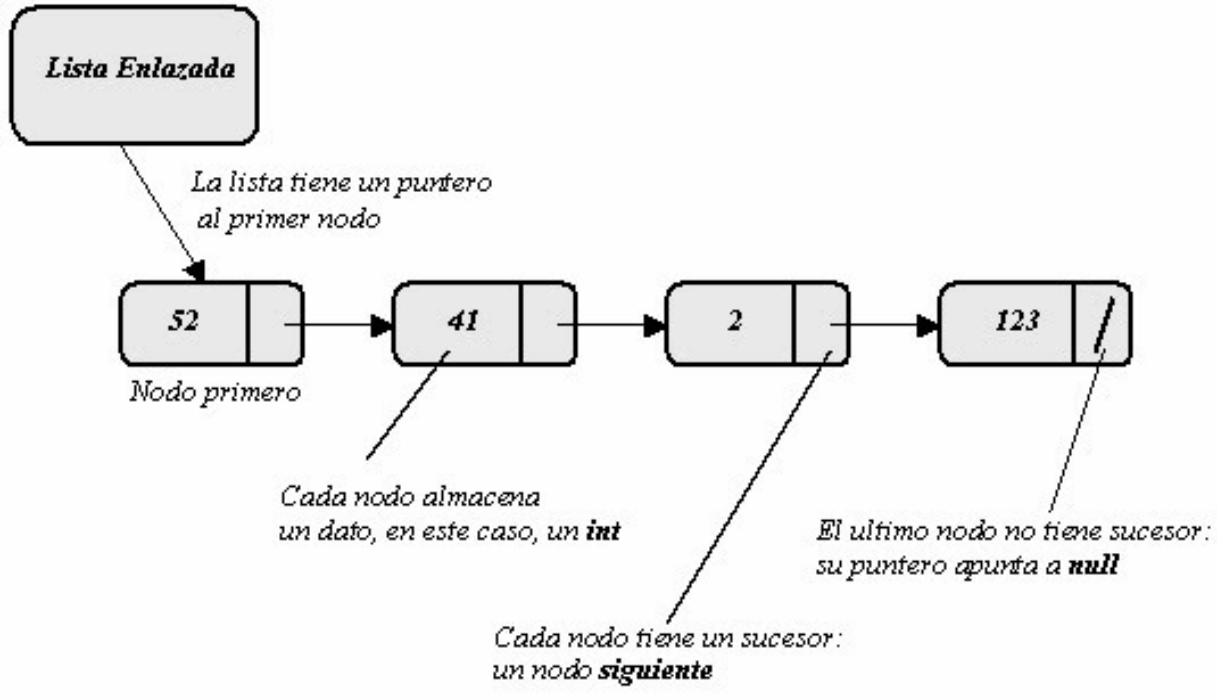
Aplicaciones de listas ligadas

Una lista enlazada es una colección lineal de elementos llamados nodos. El orden entre ellos se establece mediante punteros; direcciones o referencias a otros nodos.



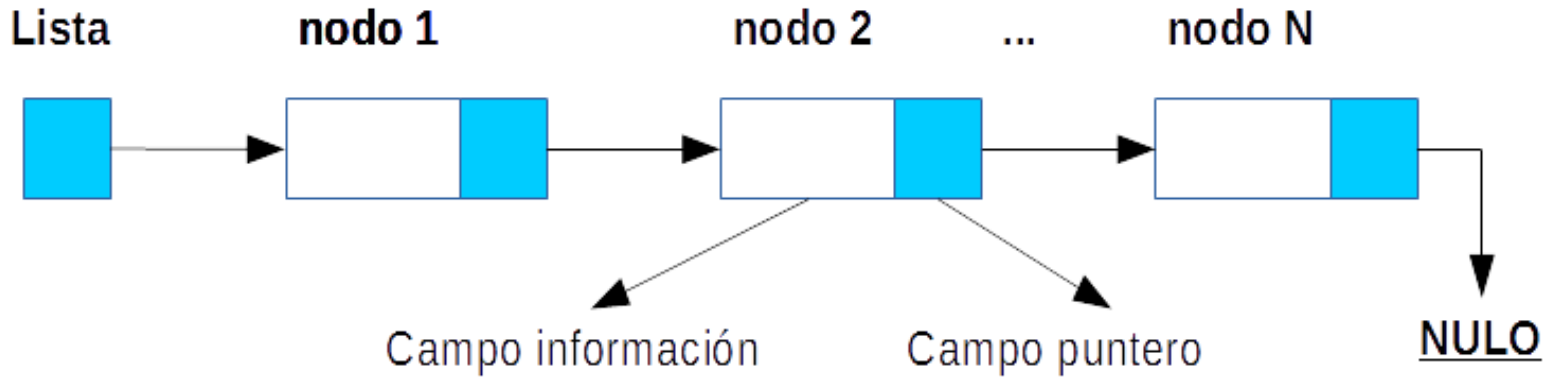
Listas ligadas o enlazadas.

- Un nodo está constituido por dos partes:
  - Un campo INFORMACIÓN: Que será del tipo de los datos que se quiera almacenar en la lista.
  - Un campo LIGA de tipo puntero, que se utiliza para establecer la liga o el enlace con otro nodo de la lista.



Listas ligadas o enlazadas.

- Un programa accede a una lista enlazada mediante un apuntador al primer nodo en la lista.
- Y se accede a cada nodo subsiguiente a través del miembro apuntador de enlace almacenado en el nodo anterior.
- El apuntador de enlace en el último nodo de una lista se establece en el valor nulo (0) para marcar el final de la lista.



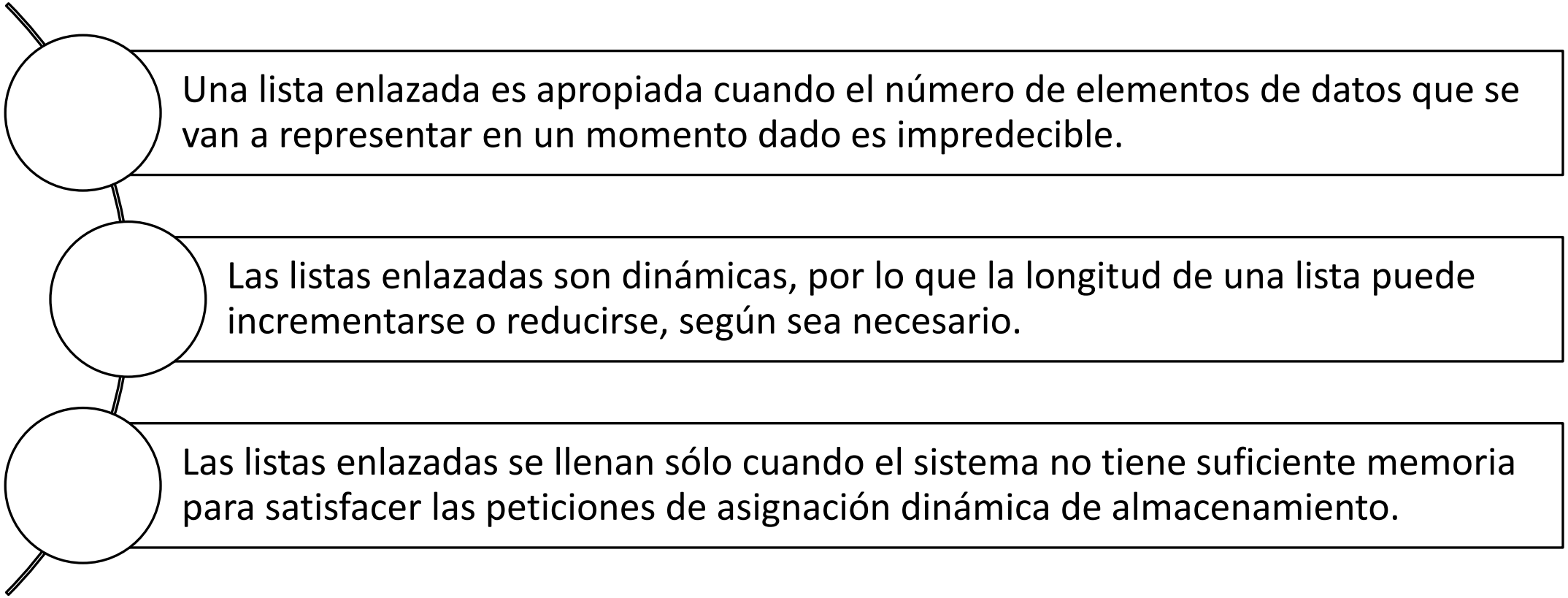
Listas ligadas o enlazadas.

Los datos se almacenan en forma dinámica en una lista enlazada; se crea cada nodo según sea necesario.

Un nodo puede contener datos de cualquier tipo, incluyendo objetos de otras clases.

Son estructuras de datos lineales.

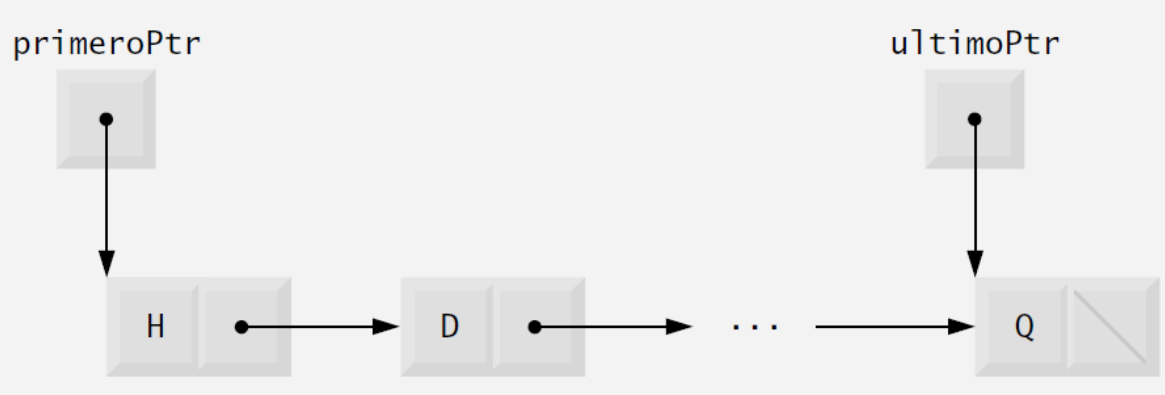
Listas ligadas o enlazadas. Ventajas.



Listas ligadas o enlazadas.

Para mantener las listas enlazadas en orden, se inserta cada nuevo elemento en el punto apropiado en la lista. Los elementos existentes de una lista no necesitan moverse.

Los nodos de las listas enlazadas no se almacenan contiguamente en memoria. Sin embargo, en sentido lógico los nodos de una lista enlazada parecen estar contiguos.



Listas ligadas o enlazadas.

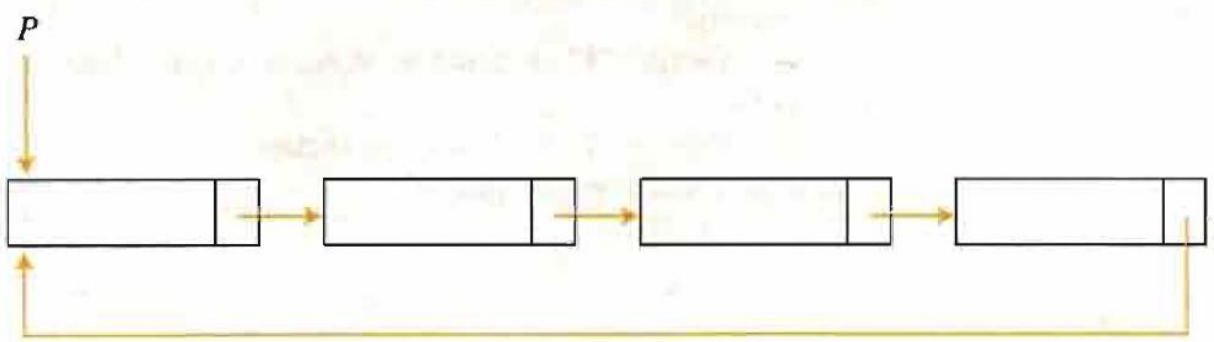
Tipos de listas ligadas

Listas simplemente ligadas

Listas doblemente ligadas. Colección de nodos en los que cada uno tiene dos apuntadores, uno a su predecesor y otro a su sucesor.

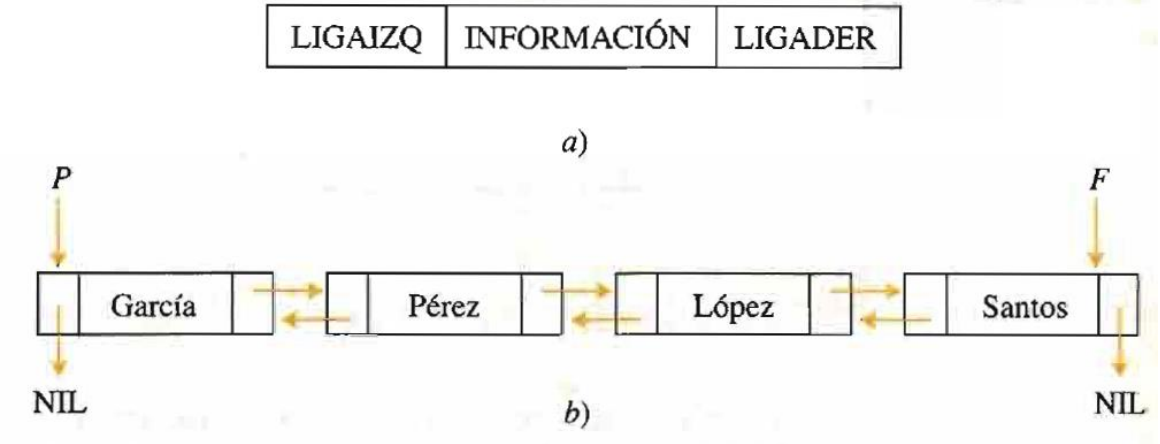
Listas circulares. El último nodo de la lista apunta al primero.

Listas ligadas o enlazadas. Ventajas.

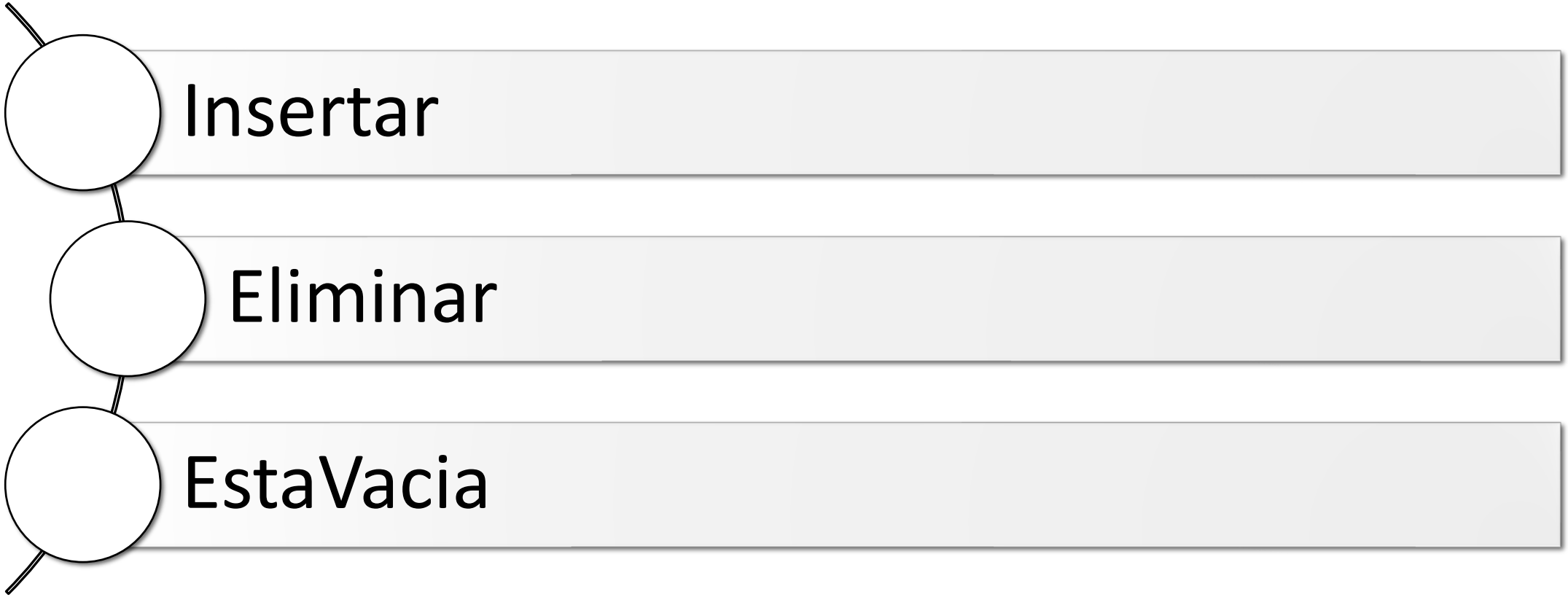


Lista ligada circular

Lista doblemente ligada



Listas ligadas o enlazadas.
Operaciones Básicas.



Listas ligadas o enlazadas.

ASIGNACIÓN DINÁMICA DE MEMORIA Y ESTRUCTURAS DE DATOS EN C++

- Para crear y mantener estructuras dinámicas de datos se requiere la asignación dinámica de memoria.
- Permite que un programa obtenga más memoria en tiempo de ejecución, para almacenar nuevos nodos.
- Cuando el programa ya no necesita la memoria, ésta se puede liberar.

Listas ligadas o enlazadas.

ASIGNACIÓN DINÁMICA DE MEMORIA Y ESTRUCTURAS DE DATOS EN C++

- El operador **new** recibe como argumento el tipo del objeto que se va a asignar en forma dinámica y devuelve un apuntador a un objeto de ese tipo. Por ejemplo:

```
Nodo *nuevoPtr = new Nodo( 10 ); // crea un Nodo con el valor 10
```

- Asigna sizeof(Nodo) bytes, ejecuta el constructor de Nodo y asigna la dirección del nuevo Nodo a nuevoPtr.
- El valor 10 se pasa al constructor de Nodo, el cual inicializa el miembro de datos del Nodo con 10.
- Si no hay memoria disponible: *bad_alloc*.

Listas ligadas o enlazadas.

ASIGNACIÓN DINÁMICA DE MEMORIA Y ESTRUCTURAS DE DATOS EN C++

- El operador **delete** ejecuta el destructor de Nodo y desasigna la memoria asignada con new; esta memoria se devuelve al sistema.

```
delete nuevoPtr;
```

- *nuevoPtr* en sí no se elimina; solo se elimina el espacio al que apunta *nuevoPtr*.
- Si el apuntador nuevoPtr tiene el valor 0 (apuntador nulo), la instrucción no tiene efecto.

Listas ligadas o enlazadas.
Operaciones Básicas.

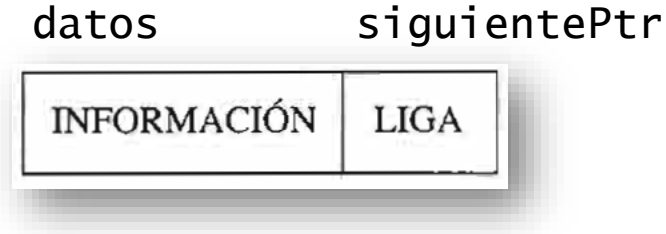
- **Insertar**
- Eliminar
- **EstaVacia**

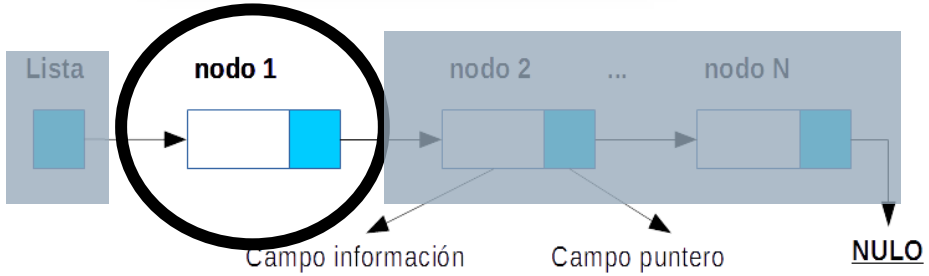
Listas ligadas o enlazadas.
Operación Insertar

- Antes de analizar la operación Insertar, se creará la clase Nodo y la clase Lista.

Clase Nodo:

```
// definicion del nodo de la lista
class Nodo
{

public:
    int datos; // INFO, variable para almacenar el
valor del nodo
    Nodo *siguientePtr;    // LIGA, apuntador al nodo
siguiente de la lista

};
```



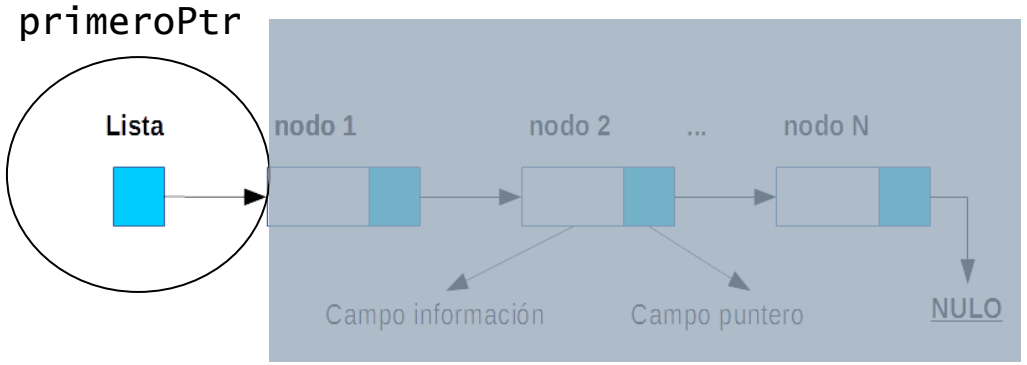


Listas ligadas o enlazadas.
Operación Insertar

- Antes de analizar la operación Insertar, se creará la clase Nodo y la clase Lista.

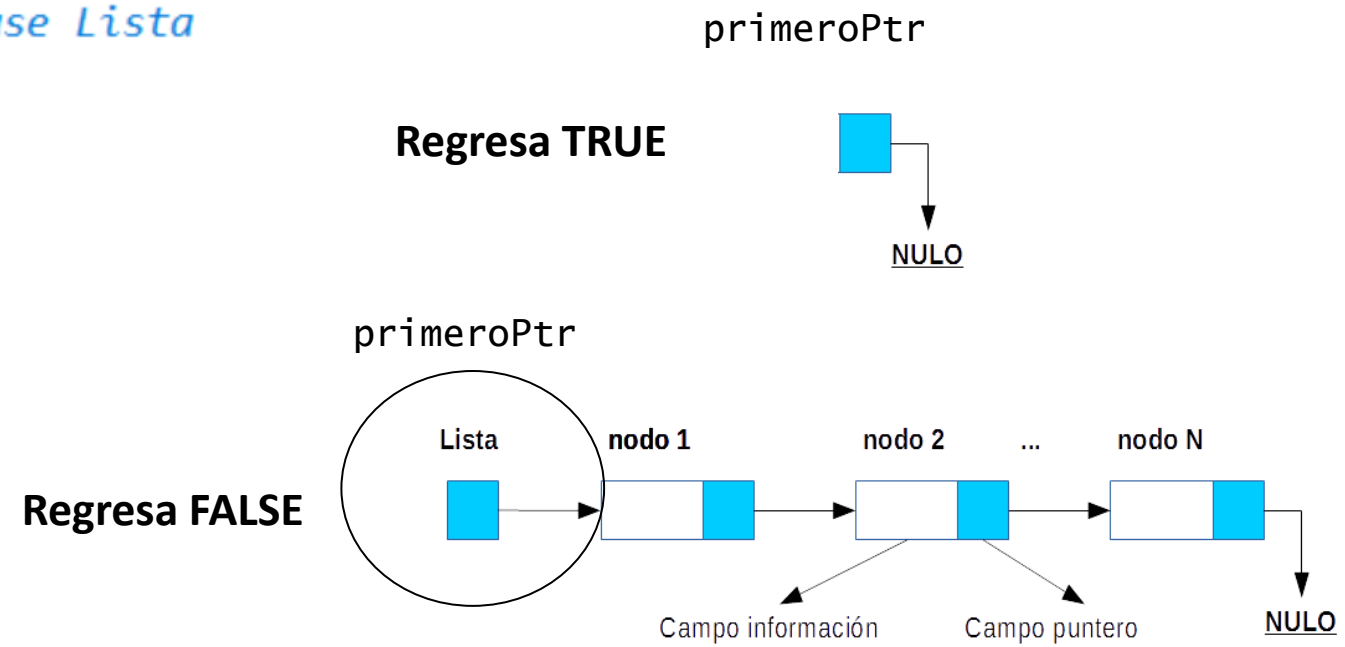
Clase Lista:

```
// definicion de la lista de nodos
class Lista
{

private:
    Nodo *primeroPtr;    // P, apuntador al inicio de la lista
    bool estaVacia();

public:
    Lista();             // constructor
    ~Lista();            // destructor
    void insertarAlInicio(int valor);       // inserta el nodo al inicio
    void recorreIterativo();    // muestra el contenido de la lista
};
```



Listas ligadas o enlazadas.
Operación Insertar

```
// definicion de funciones de la clase Lista

// verifica si la lista esta vacia
bool Lista::estaVacia()
{
    return primeroPtr == NULL;
}
```

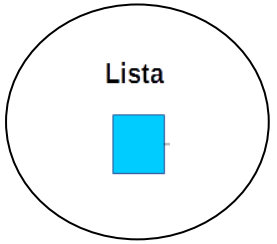


Listas ligadas o enlazadas.
Operación Insertar

- Constructor. Asigna Nulo a primeroPtr

```
// constructor predeterminado
Lista::Lista()
{
    primeroPtr = NULL;
}
```

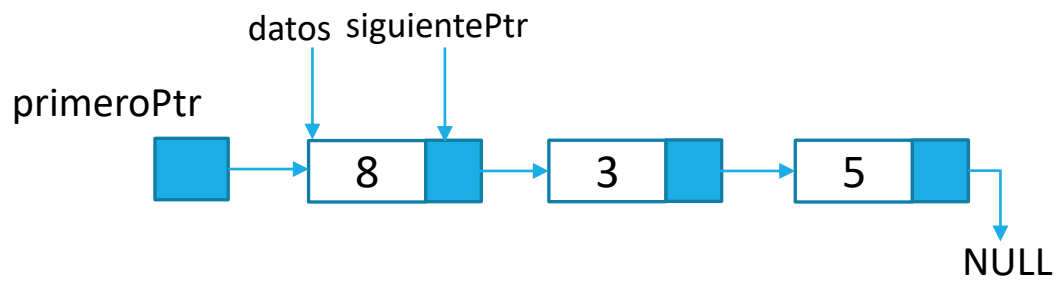
primeroPtr=NULL

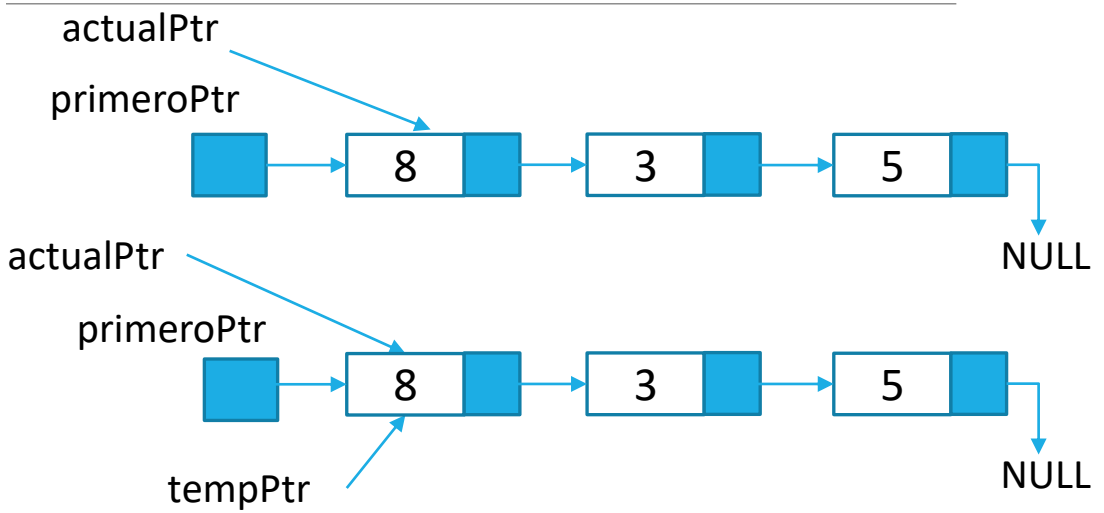


Listas ligadas o enlazadas.

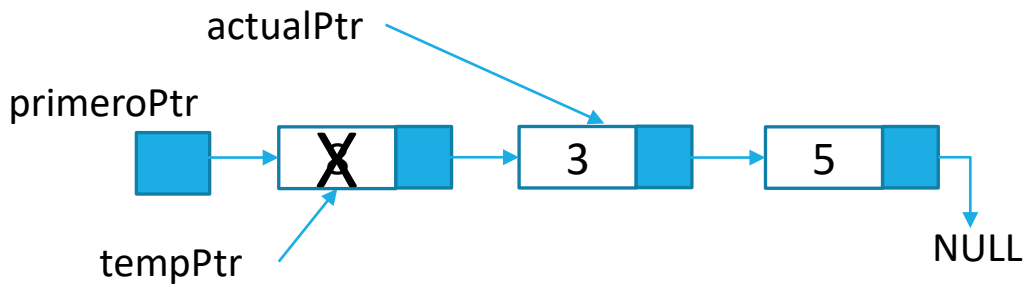
- Destructor. Para liberar la memoria una vez que finaliza el programa.

```
// destructor predeterminado
Lista::~Lista()
{
    if( !estaVacia() )
    {
        cout << "\n\nDestruyendo nodos... \n\n";

        Nodo *actualPtr = primeroPtr;
        Nodo *tempPtr;

        // elimina los nodos restantes
        while( actualPtr != NULL )
        {
            tempPtr = actualPtr;
            cout << tempPtr->datos << ' ';
            actualPtr = actualPtr->siguientePtr;
            delete tempPtr;
        }
    }

    cout << "\n\nSe destruyeron todos los nodos\n\n";
}
```





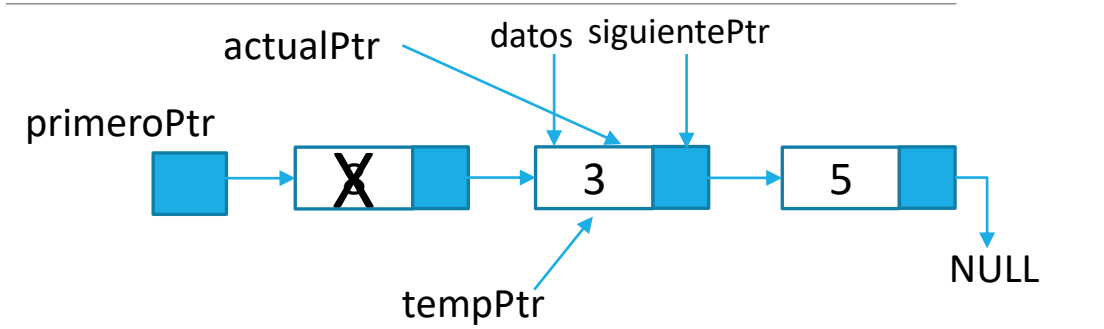
Imprime 8



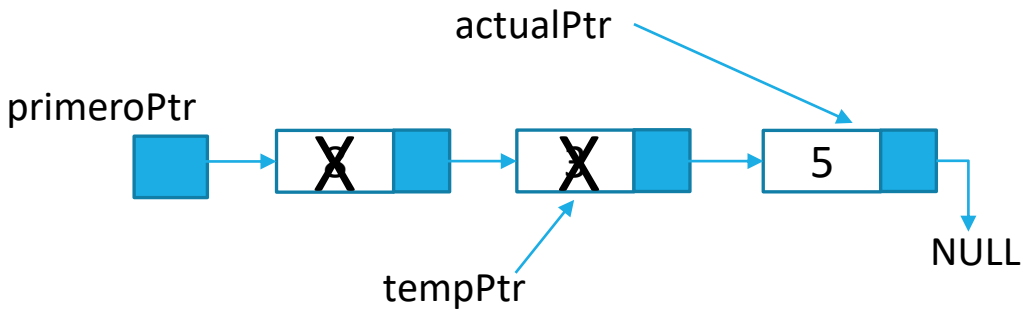
Listas ligadas o enlazadas.

- Destructor. Para liberar la memoria una vez que finaliza el programa.

```
// destructor predeterminado
Lista::~Lista()
{
    if( !estaVacia() )
    {
        cout << "\n\nDestruyendo nodos... \n\n";

        Nodo *actualPtr = primeroPtr;
        Nodo *tempPtr;

        // elimina los nodos restantes
        while( actualPtr != NULL )
        {
            tempPtr = actualPtr;
            cout << tempPtr->datos << ' ';
            actualPtr = actualPtr->siguientePtr;
            delete tempPtr;
        }
    }

    cout << "\n\nSe destruyeron todos los nodos\n\n";
}
```



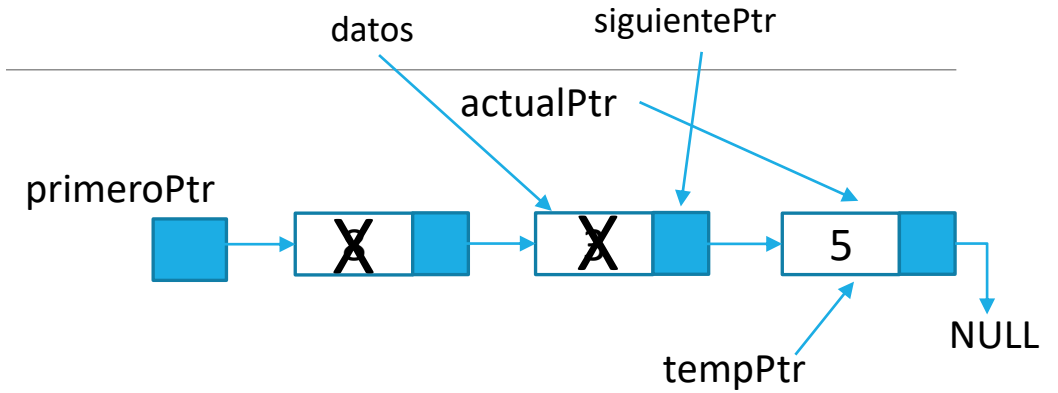
Imprime 3



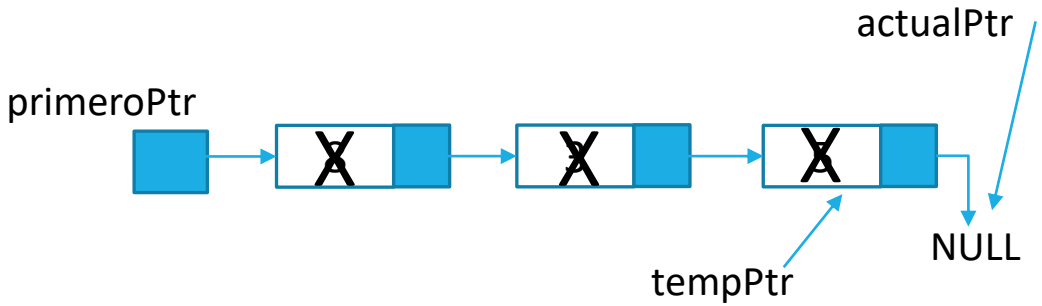
Listas ligadas o enlazadas.

- Destructor. Para liberar la memoria una vez que finaliza el programa.

```
// destructor predeterminado
Lista::~Lista()
{
    if( !estaVacia() )
    {
        cout << "\n\nDestruyendo nodos... \n\n";

        Nodo *actualPtr = primeroPtr;
        Nodo *tempPtr;

        // elimina los nodos restantes
        while( actualPtr != NULL )
        {
            tempPtr = actualPtr;
            cout << tempPtr->datos << ' ';
            actualPtr = actualPtr->siguientePtr;
            delete tempPtr;
        }
    }

    cout << "\n\nSe destruyeron todos los nodos\n\n";
}
```



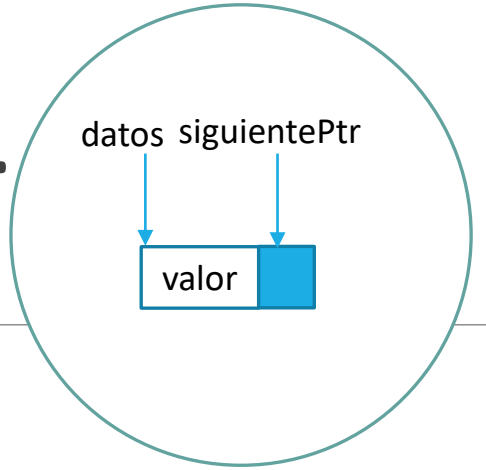
Imprime 5

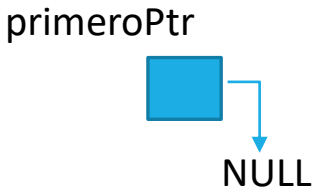


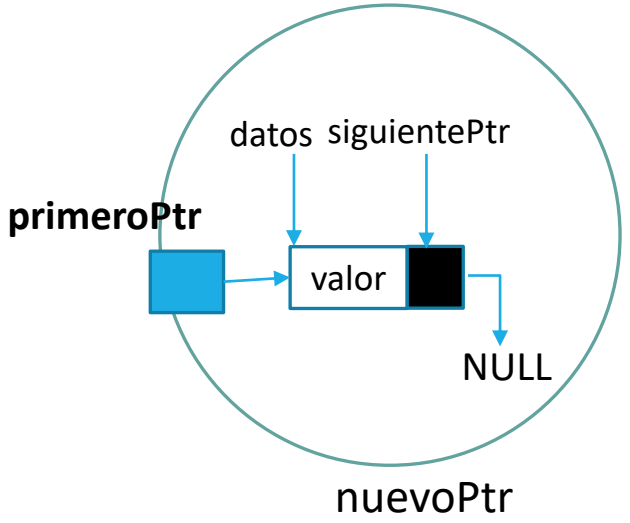
Listas ligadas o enlazadas.
Operación Insertar

- Insertar al inicio

```
void Lista::insertarAlInicio(int valor)
{
    Nodo *nuevoPtr = new Nodo();     // Q, var
    nuevoPtr->datos = valor;         // DATO,

    if( estaVacia() )   // la lista esta vaci
    {
        nuevoPtr->siguientePtr = NULL;  // el
    }
    else    // la lista no esta vacía
    {
        nuevoPtr->siguientePtr = primeroPtr;
    }

    primeroPtr = nuevoPtr;  // apunta el prim
}
```



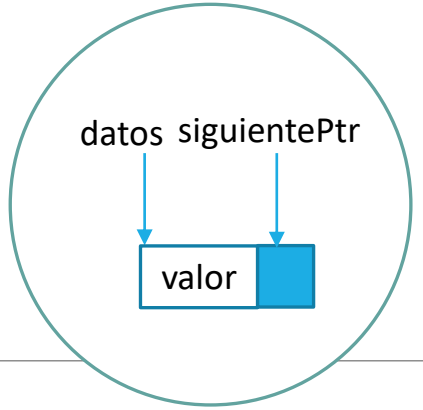


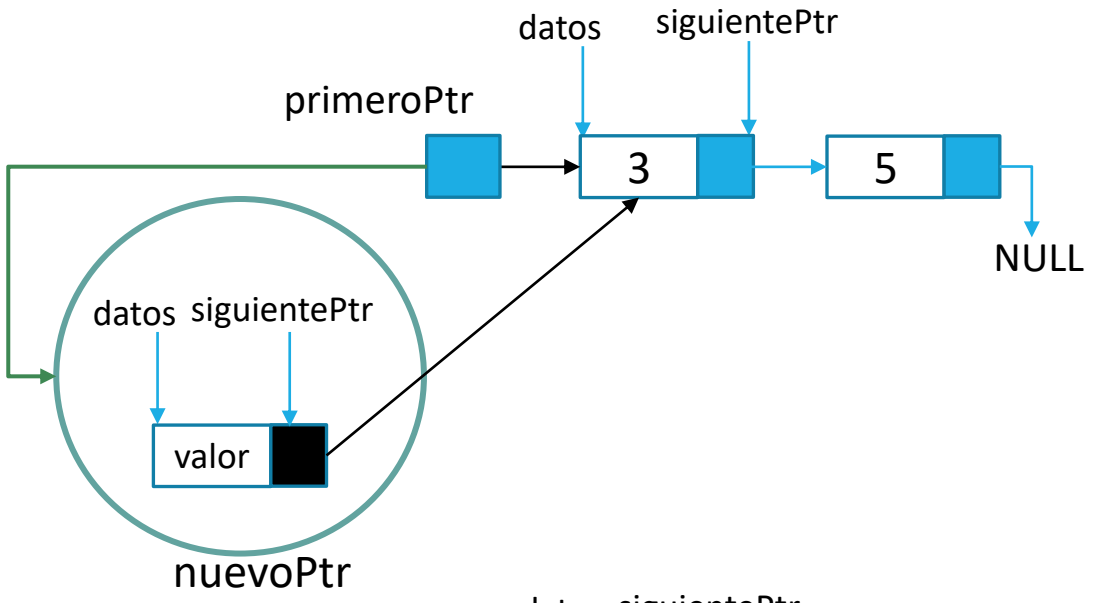


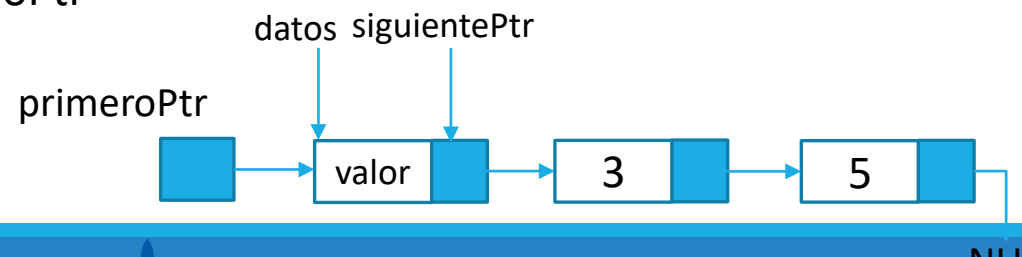
Listas ligadas o enlazadas.
Operación Insertar

- Insertar al inicio

```
void Lista::insertarAlInicio(int valor)
{
    Nodo *nuevoPtr = new Nodo();     // Q, var
    nuevoPtr->datos = valor;         // DATO,

    if( estaVacia() )   // la lista esta vaci
    {
        nuevoPtr->siguientePtr = NULL;  // el
    }
    else    // la lista no esta vacía
    {
        nuevoPtr->siguientePtr = primeroPtr;
    }

    primeroPtr = nuevoPtr;  // apunta el prim
}
```







Listas ligadas o enlazadas.

- Actividad:

- Copia en C++ el siguiente código correspondiente a cada una de las clases y métodos estudiados anteriormente para manipular las Listas Enlazadas.

```cpp
#include <iostream>
using namespace std;

// definicion del nodo de la lista
class Nodo
{

public:
    int datos;  // INFO, variable para almacenar el valor del nodo
    Nodo *siguientePtr; // LIGA, apuntador al nodo siguiente de la lista

};

// definicion de la lista de nodos
class Lista
{

private:
    Nodo *primeroPtr;   // P, apuntador al inicio de la lista
    bool estaVacia();

public:
    Lista();            // constructor
    ~Lista();           // destructor
    void insertarAlInicio(int valor);       // inserta el nodo al inicio
    void recorreIterativo();    // muestra el contenido de la lista
};

// definicion de funciones de la clase Lista
```

```
28
29  // definicion de funciones de la clase Lista
30
31  // verifica si la lista esta vacia
32  bool Lista::estaVacia()
33  {
34      return primeroPtr == NULL;
35  }
36
37  // constructor predeterminado
38  Lista::Lista()
39  {
40      primeroPtr = NULL;
41  }
42
43  // destructor predeterminado
44  Lista::~Lista()
45  {
46      if( !estaVacia() )
47      {
48          cout << "\n\nDestruyendo nodos... \n\n";
49
50          Nodo *actualPtr = primeroPtr;
51          Nodo *tempPtr;
52
53          // elimina los nodos restantes
```

```
52
53          // elimina los nodos restantes
54          while( actualPtr != NULL )
55          {
56              tempPtr = actualPtr;
57              cout << tempPtr->datos << ' ';
58              actualPtr = actualPtr->siguientePtr;
59              delete tempPtr;
60          }
61      }
62
63      cout << "\n\nSe destruyeron todos los nodos\n\n";
64  }
65
66  void Lista::insertarAlInicio(int valor)
67  {
68      Nodo *nuevoPtr = new Nodo();     // Q, variable nodo temporal
69      nuevoPtr->datos = valor;
70
71      if( estaVacia() )   // la lista esta vacia
72      {
73          nuevoPtr->siguientePtr = NULL;  // el nuevo nodo apunta a nulo
74      }
75      else    // la lista no esta vacía
76      {
77          nuevoPtr->siguientePtr = primeroPtr;    // apunta el nuevo nodo al nodo que antes era el primero
78      }
79
80      primeroPtr  = nuevoPtr; // apunta el primer nodo de la lista hacia el nuevo nodo
```

```
79
80          primeroPtr  = nuevoPtr; // apunta el primer nodo de la lista hacia el nuevo nodo
81      }
82
83      void Lista::recorreIterativo()
84      {
85          if( estaVacia() )   // la lista esta vacia
86          {
87              cout << "\nLa lista esta vacia\n\n";
88              system("pause");
89              return;
90          }
91
92          Nodo *actualPtr = primeroPtr;
93
94          cout << "\nLos elementos de la lista son: ";
95
96          while( actualPtr != NULL )  // obtiene los datos del elemento
97          {
98              cout << actualPtr->datos << " -> ";
99              actualPtr = actualPtr->siguientePtr;
100         }
101
102         cout << "\n\n";
103         system("pause");
104     }
105
106     // definicion de funciones de la clase Lista
```

```
106  // definicion de funciones de la clase Lista
107
108  // imprime las instrucciones
109  void menu()
110  {
111      system("cls");
112      cout << "\n  ..[ LISTA SIMPLEMENTE LIGADA ]..";
113      cout << "\n  ..[ Erika Meneses Rico ].. \n\n";
114      cout << "[1] Insertar elemento al inicio \n";
115      cout << "[2] Imprimir los valores de la lista \n";
116      cout << "[3] SALIR \n";
117      cout << "\nIngrese opcion : ";
118  }
119
120  // rutina principal
121  int main()
122  {
123      int opcion; // almacena la opcion del usuario
124      int valor;  // almacena el valor del nodo
125
126      Lista listaEnteros; // crea el objeto Lista
127
128      system("color 1F");   // poner color a la consola
129
130      do
131      {
132          menu();
133          cin >> opcion;
134
```

```
126     Lista listaEnteros; // crea el objeto Lista
127
128     system("color 1F");   // poner color a la consola
129
130     do
131     {
132         menu();
133         cin >> opcion;
134
135         switch( opcion )
136         {
137             case 1: // inserta al principio
138                 cout << "\nIngrese valor entero: ";
139                 cin >> valor;
140                 listaEnteros.insertarAlInicio( valor );
141                 listaEnteros.recorreIterativo();
142                 break;
143             case 2: // imprime la lista
144                 listaEnteros.recorreIterativo();
145                 break;
146         }
147     }while( opcion != 3 );
148
149     return 0;
150 }
```

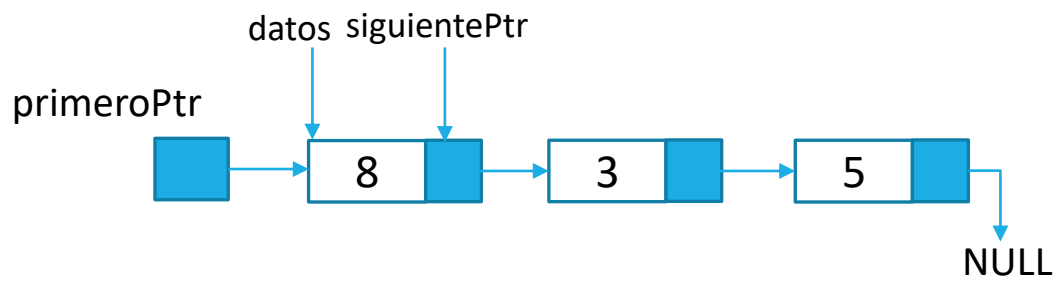
Listas ligadas o enlazadas.
Operaciones Complementarias.

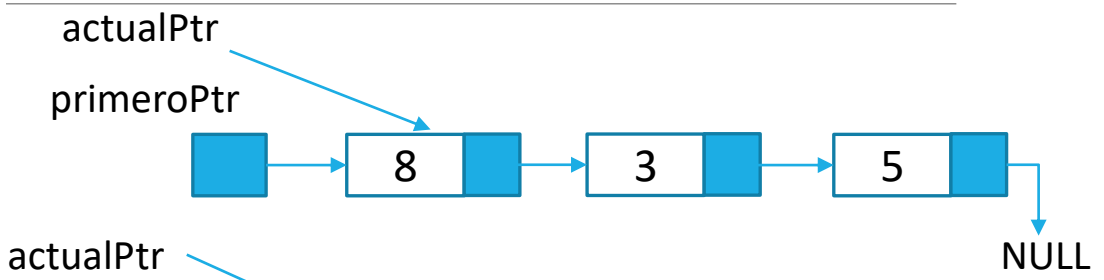
- Recorre Iterativo
- Inserta al Final
- Recorre Recursivo
- Inserta Antes

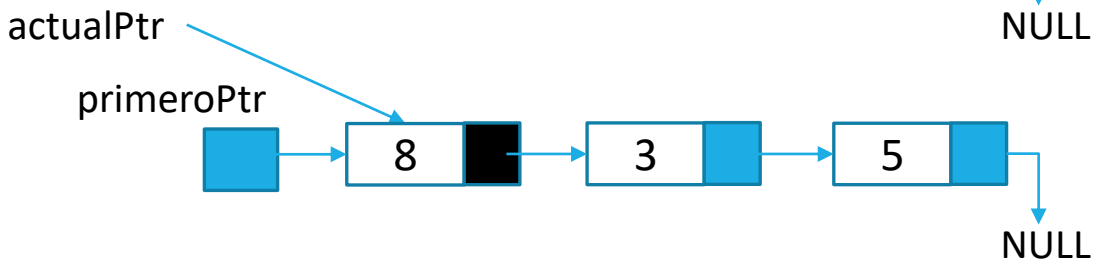
Listas ligadas o enlazadas.

- Recorre Iterativo

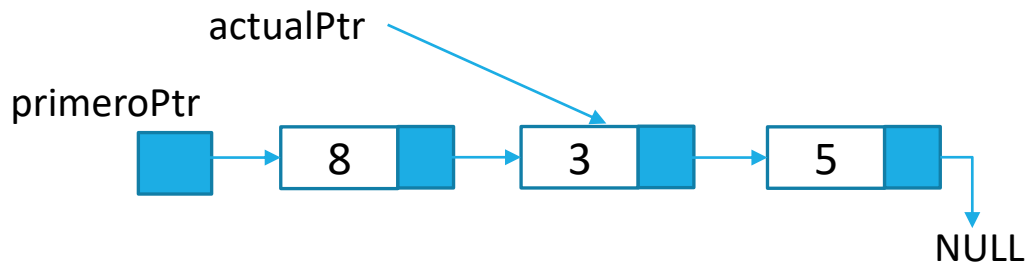
```
void Lista::recorreIterativo()
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\nLa lista esta vacia\n\n";
        system("pause");
        return;
    }

    Nodo *actualPtr = primeroPtr;

    cout << "\nLos elementos de la lista son: ";

    while( actualPtr != NULL )  // obtiene los datos del elemento
    {
        cout << actualPtr->datos << " -> ";
        actualPtr = actualPtr->siguientePtr;
    }

    cout << "\n\n";
    system("pause");
}
```







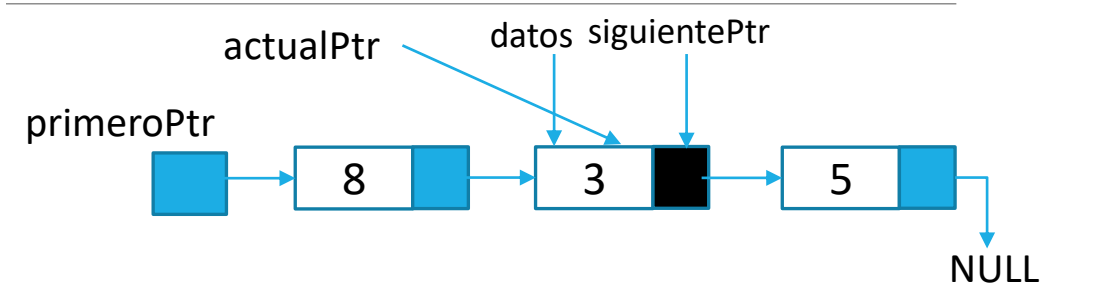
Imprime 8



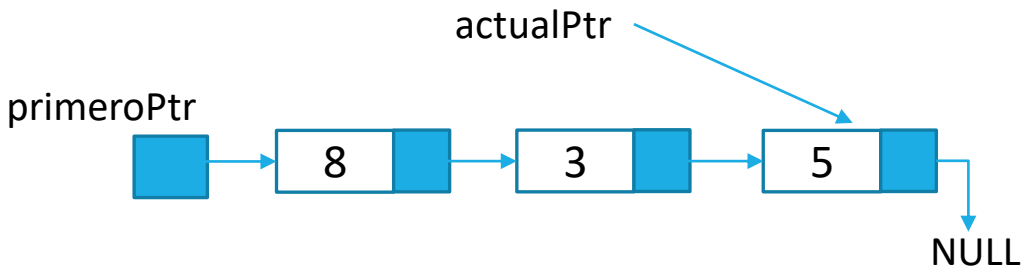
Listas ligadas o enlazadas.

- Recorre Iterativo

```
void Lista::recorreIterativo()
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\nLa lista esta vacia\n\n";
        system("pause");
        return;
    }

    Nodo *actualPtr = primeroPtr;

    cout << "\nLos elementos de la lista son: ";

    while( actualPtr != NULL )  // obtiene los datos del elemento
    {
        cout << actualPtr->datos << " -> ";
        actualPtr = actualPtr->siguientePtr;
    }

    cout << "\n\n";
    system("pause");
}
```



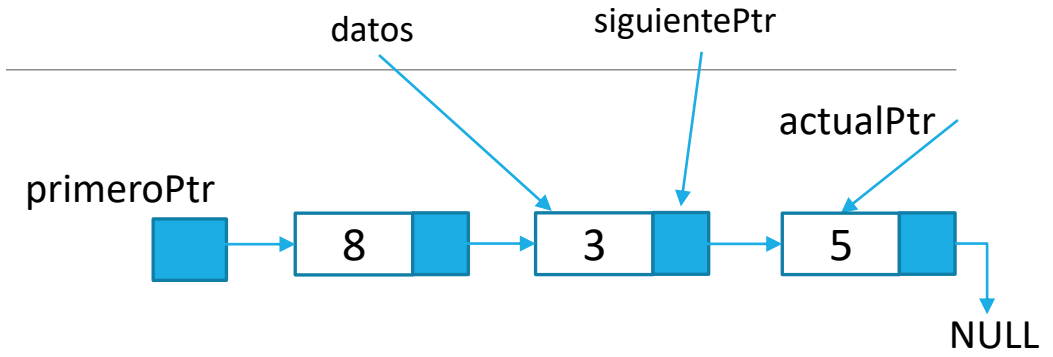
Imprime 3



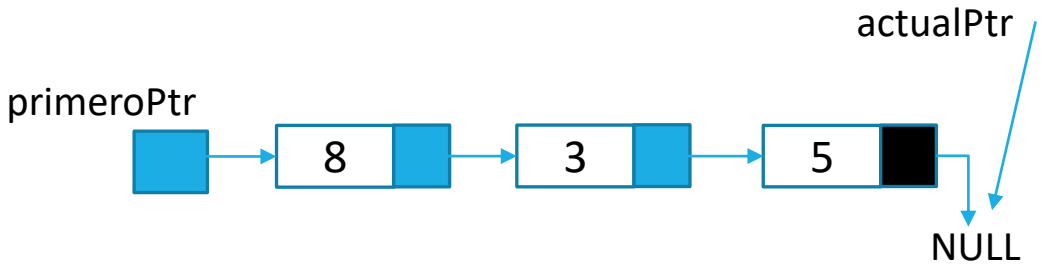
Listas ligadas o enlazadas.

- Recorre Iterativo

```
void Lista::recorreIterativo()
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\nLa lista esta vacia\n\n";
        system("pause");
        return;
    }

    Nodo *actualPtr = primeroPtr;

    cout << "\nLos elementos de la lista son: ";

    while( actualPtr != NULL )  // obtiene los datos del elemento
    {
        cout << actualPtr->datos << " -> ";
        actualPtr = actualPtr->siguientePtr;
    }

    cout << "\n\n";
    system("pause");
}
```



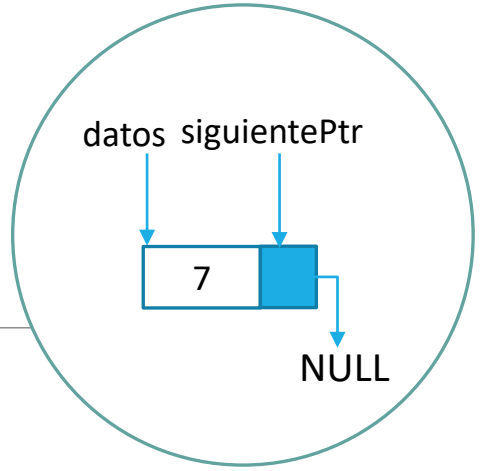
Imprime 5



Listas ligadas o enlazadas.

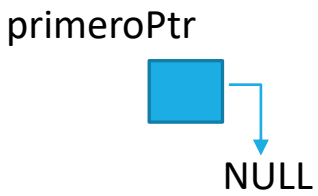
- Inserta al Final

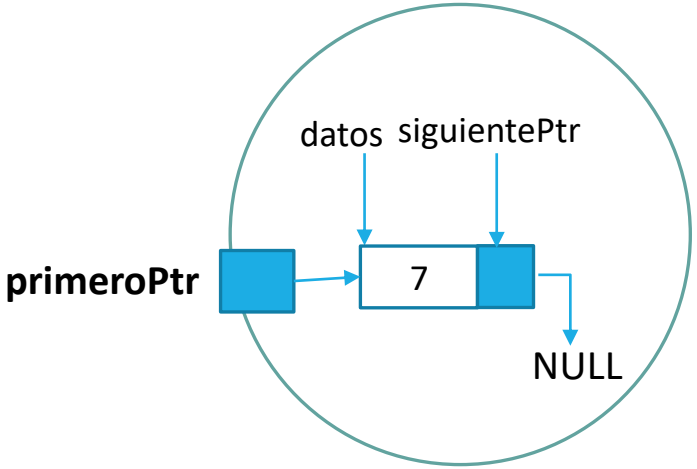
```
void Lista::insertarAlFinal(int valor)
{
    Nodo *nuevoPtr = new Nodo();     // Q, variable n
    nuevoPtr->datos = valor;         // DATO, se guar
    nuevoPtr->siguientePtr = NULL;   // el nuevo nodo

    if( estaVacia() )   // la lista esta vacia
    {
        primeroPtr = nuevoPtr;  // apunta el primer
    }
    else    // la lista no esta vacía
    {
        Nodo *actualPtr = primeroPtr;

        while( actualPtr->siguientePtr != NULL )
        {
            actualPtr = actualPtr->siguientePtr;
        }
        actualPtr->siguientePtr = nuevoPtr; // el ul
    }
}
```



nuevoPtr

Si estaVacia()



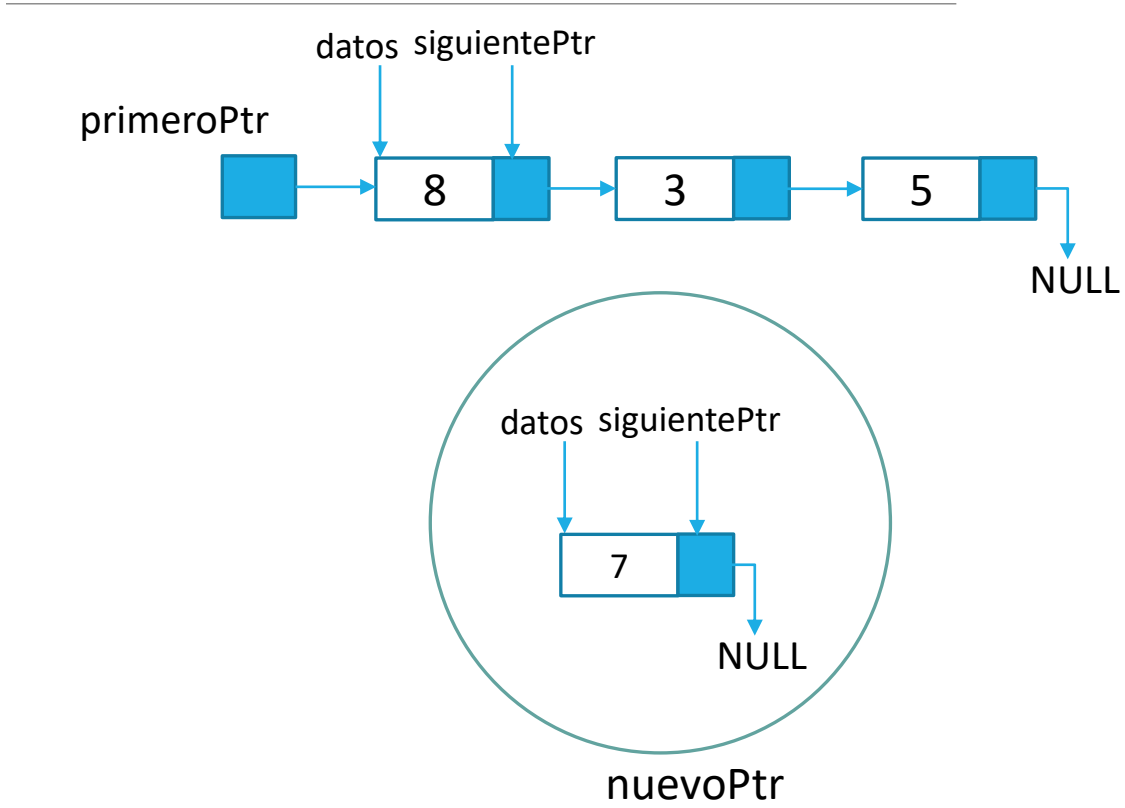


nuevoPtr

Listas ligadas o enlazadas.

- Inserta al Final

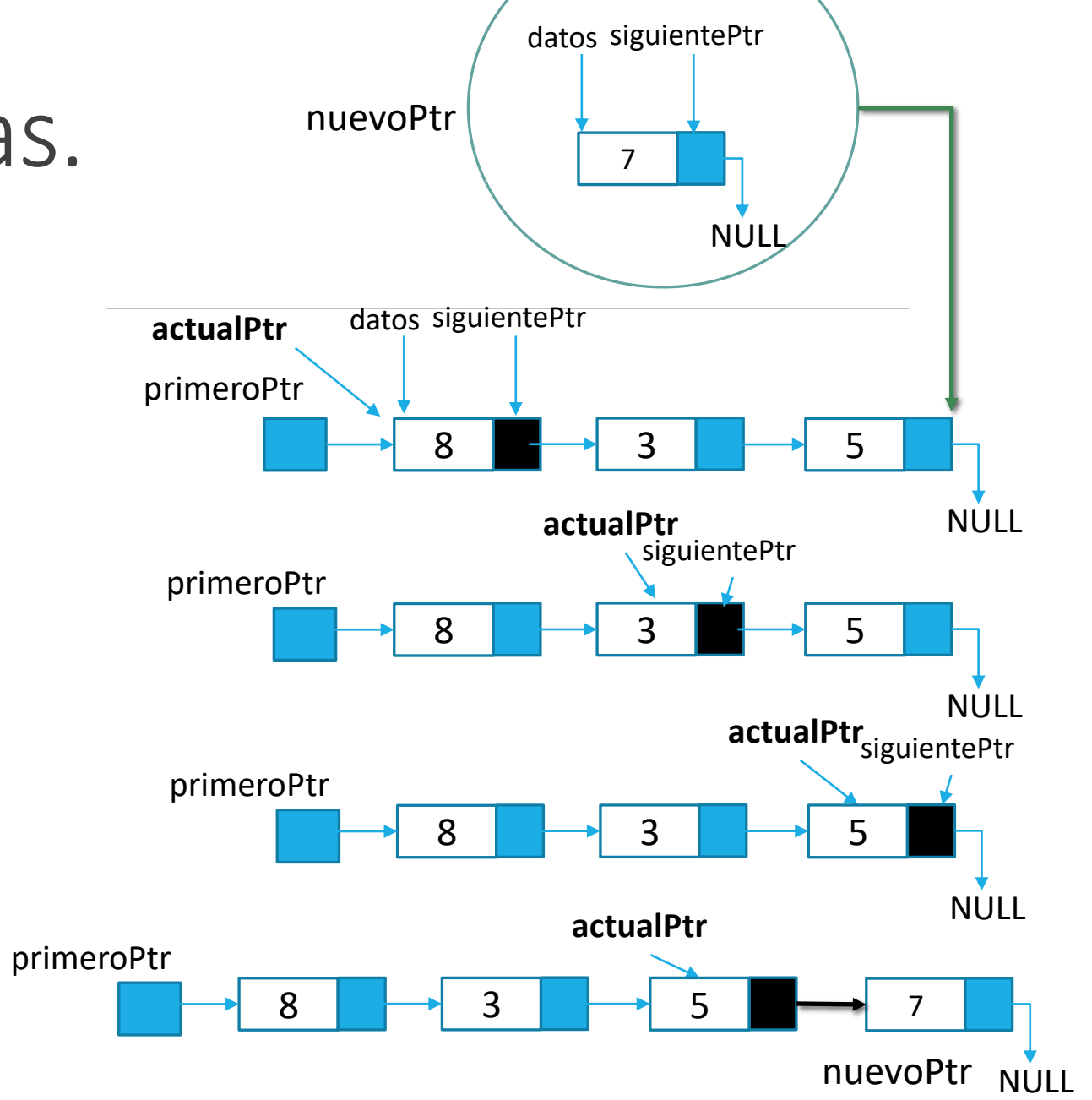
```
void Lista::insertarAlFinal(int valor)
{
    Nodo *nuevoPtr = new Nodo();     // Q, variable n
    nuevoPtr->datos = valor;         // DATO, se guar
    nuevoPtr->siguientePtr = NULL;   // el nuevo nodo

    if( estaVacia() )   // la lista esta vacia
    {
        primeroPtr = nuevoPtr;  // apunta el primer
    }
    else    // la lista no esta vacía
    {
        Nodo *actualPtr = primeroPtr;

        while( actualPtr->siguientePtr != NULL )
        {
            actualPtr = actualPtr->siguientePtr;
        }
        actualPtr->siguientePtr = nuevoPtr; // el ul
    }
}
```



Listas ligadas o enlazadas.

- Inserta al Final

```
void Lista::insertarAlFinal(int valor)
{
    Nodo *nuevoPtr = new Nodo();     // Q, variable n
    nuevoPtr->datos = valor;         // DATO, se guar
    nuevoPtr->siguientePtr = NULL;   // el nuevo nodo

    if( estaVacia() )   // la lista esta vacia
    {
        primeroPtr = nuevoPtr;  // apunta el primer
    }
    else    // la lista no esta vacía
    {
        Nodo *actualPtr = primeroPtr;

        while( actualPtr->siguientePtr != NULL )
        {
            actualPtr = actualPtr->siguientePtr;
        }
        actualPtr->siguientePtr = nuevoPtr; // el ul
    }
}
```



Listas ligadas o enlazadas.

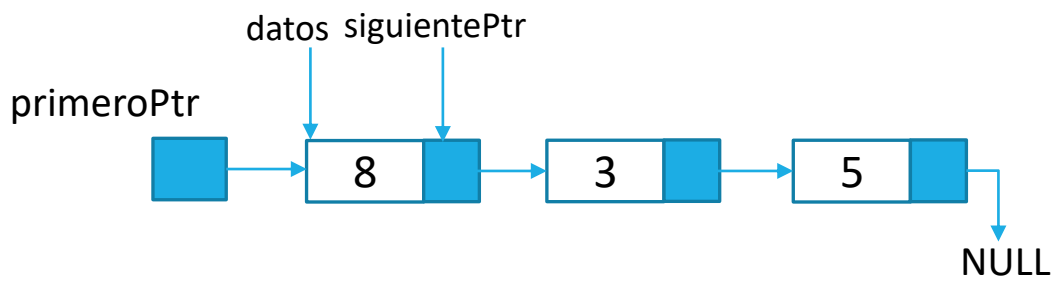
- Llamado a Recorre Recursivo

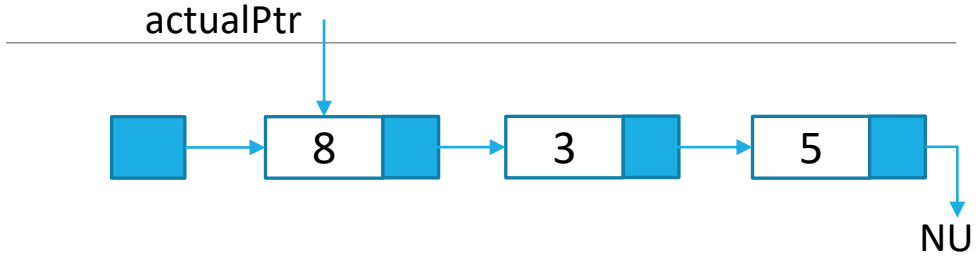
```
void Lista::llamadoRecorreRecursivo()
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\nLa lista esta vacia\n\n";
        system("pause");
        return;
    }

    cout << "\nLos elementos de la lista recursivo son: ";

    recorreRecursivo( primeroPtr ); // funcion recursiva

    cout << "\n\n";
    system("pause");
}
```

Listas ligadas o enlazadas.

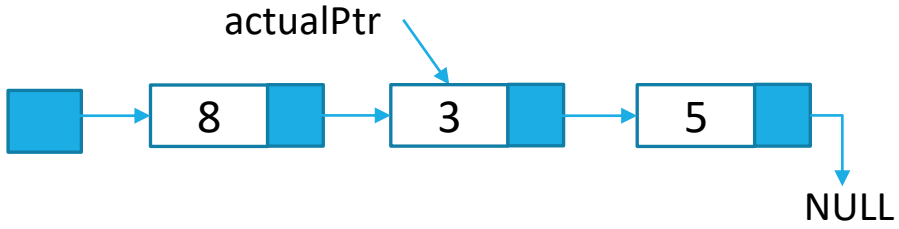
- Recorre Recursivo

```
// recorre una lista en forma recursiva
void Lista::recorreRecursivo( Nodo *actualPtr )
{
    if( actualPtr != NULL ) // obtiene los datos del
    {
        cout << actualPtr->datos << ' ';
        recorreRecursivo( actualPtr->siguientePtr );
    }
}
```

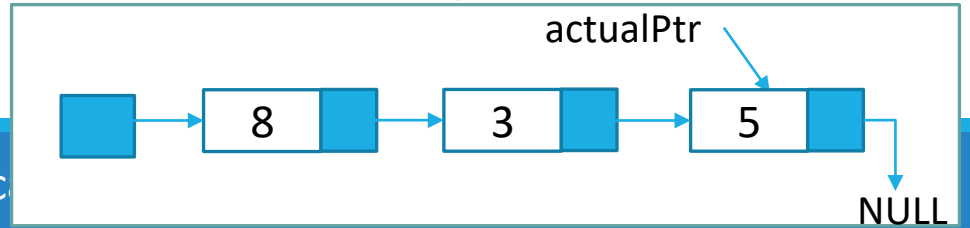




Imprime 8



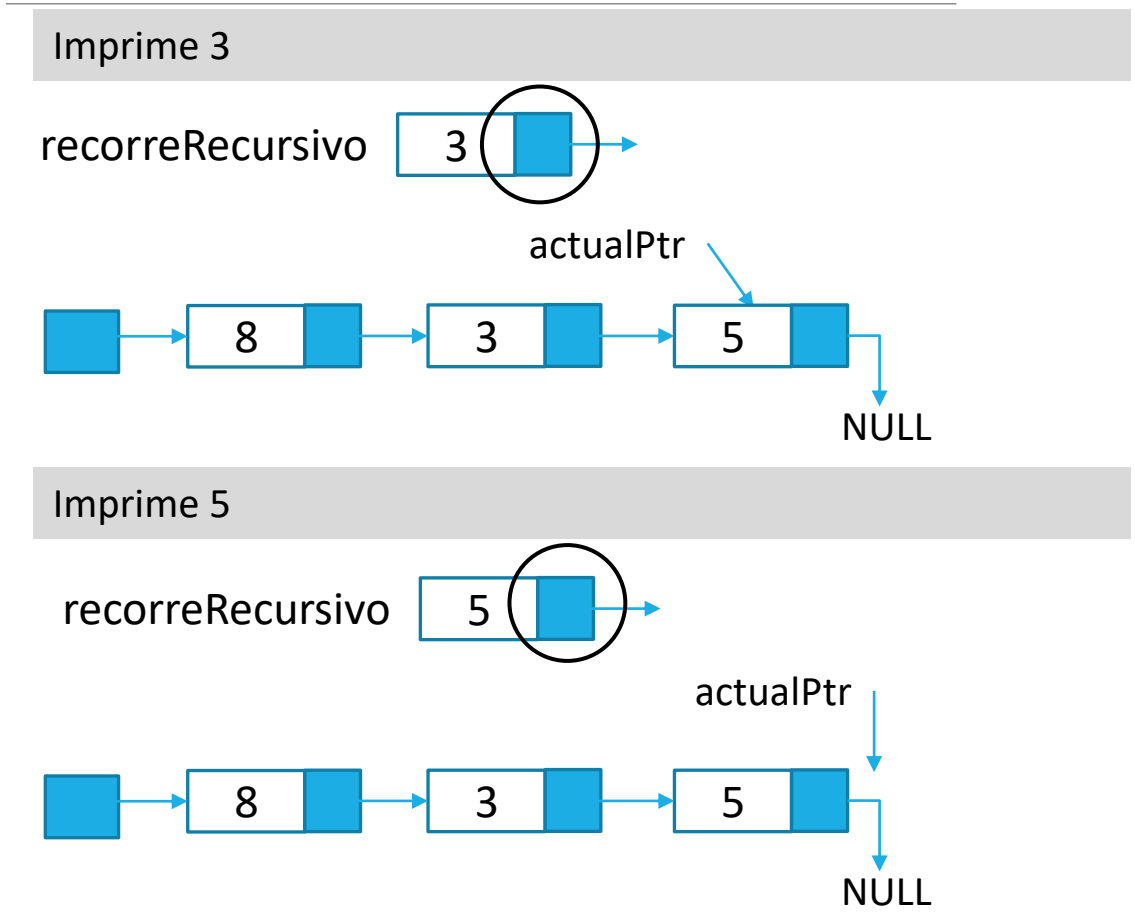
Imprime 3



Listas ligadas o enlazadas.

- Recorre Recursivo

```
// recorre una lista en forma recursiva
void Lista::recorreRecursivo( Nodo *actualPtr )
{
    if( actualPtr != NULL ) // obtiene los datos del
    {
        cout << actualPtr->datos << ' ';
        recorreRecursivo( actualPtr->siguientePtr );
    }
}
```



Listas ligadas o enlazadas. ○ Insertar Antes

```
void Lista::insertarAntes(int referencia, int valor)
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\n\nEl nodo dado como referencia no se encuentra en la lista. ";
        system("pause");
        return;
    }

    Nodo *nuevoPtr = new Nodo();     // Q, variable nodo temporal
    nuevoPtr->datos = valor;         // DATO, se guarda el nuevo valor
    nuevoPtr->siguientePtr = NULL;   // el nuevo nodo apunta a nulo

    Nodo *actualPtr = primeroPtr;    // obtiene el primer nodo de la lista
    Nodo *previoPtr;                 // variable que guarda el nodo anterior al actual
    int BAND = 1;                    // variable que indica si se encontró o no el valor

    while( actualPtr->datos != referencia && BAND == 1 )    // recorre hasta encontrar el valor o llegar al último nodo
    {
        if( actualPtr->siguientePtr != NULL )
        {
            previoPtr = actualPtr;  // guarda el nodo actual antes de saltar al siguiente
            actualPtr = actualPtr->siguientePtr;
        }
        else    // se llegó al final y no se encontro
        {
            BAND = 0;
        }
    }
```

Si la lista está vacía

Crea nuevo nodo con el valor a insertar

Variables temporales

Busca el valor de referencia

Listas ligadas o enlazadas. ○ Insertar Antes

```
if( BAND == 1 ) // si se encontro el nodo de referencia
{
    if( primeroPtr == actualPtr )   // si el nodo de referencia es el primero
    {
        nuevoPtr->siguientePtr = primeroPtr;
        primeroPtr = nuevoPtr;      // el nuevo nodo es el primero
    }
    else
    {
        previoPtr->siguientePtr = nuevoPtr; // el nodo previo al actual apunta al nuevo nodo
        nuevoPtr->siguientePtr = actualPtr; // el nodo nuevo, apunta al nodo de referencia
    }
}
else
{
    cout << "\n\nEl nodo dado como referencia no se encuentra en la lista. ";
}

}
```

Inserta el nodo haciendo los cambios correspondientes el apuntadores

Listas ligadas o enlazadas.

- Insertar Antes

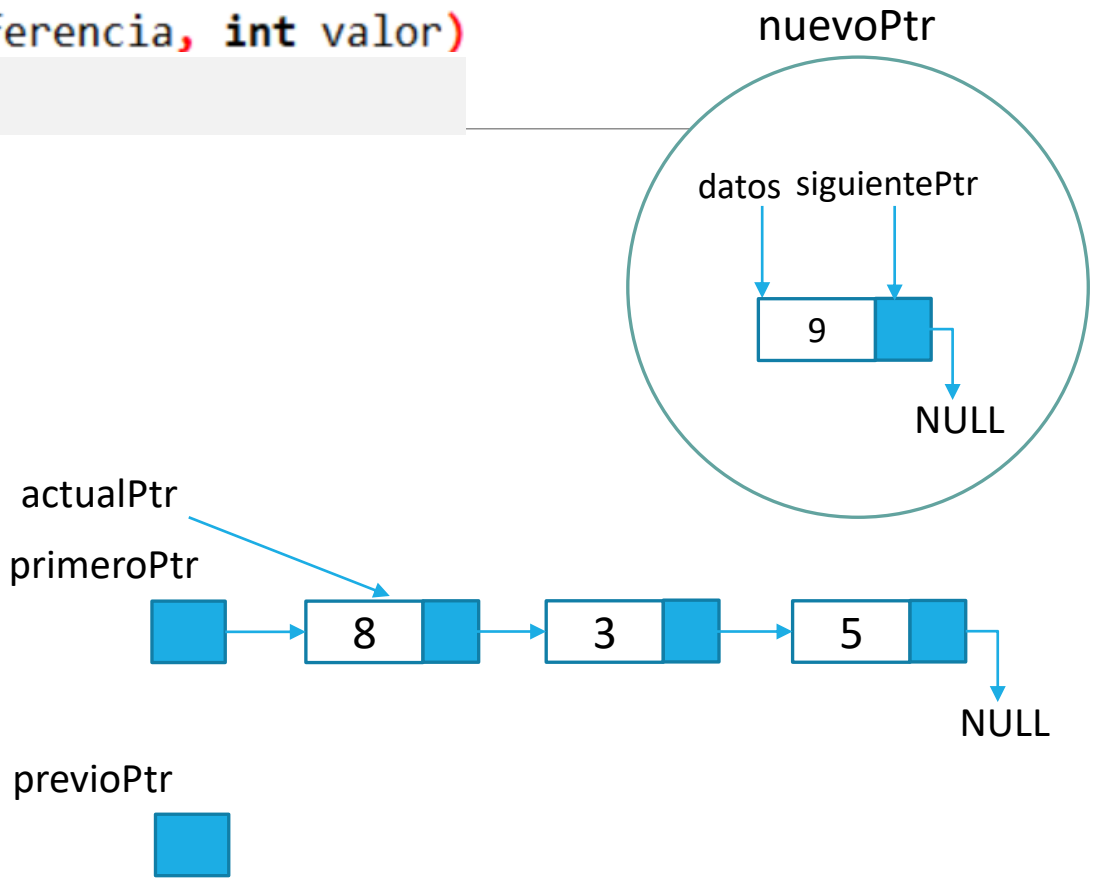
```
void Lista::insertarAntes(int referencia, int valor)
```

referencia = **5**, valor = **9**

```
Nodo *nuevoPtr = new Nodo();
nuevoPtr->datos = valor;
nuevoPtr->siguientePtr = NULL;

Nodo *actualPtr = primeroPtr;
Nodo *previoPtr;
int BAND = 1;
```

Crea nuevo nodo

Varia bles temp

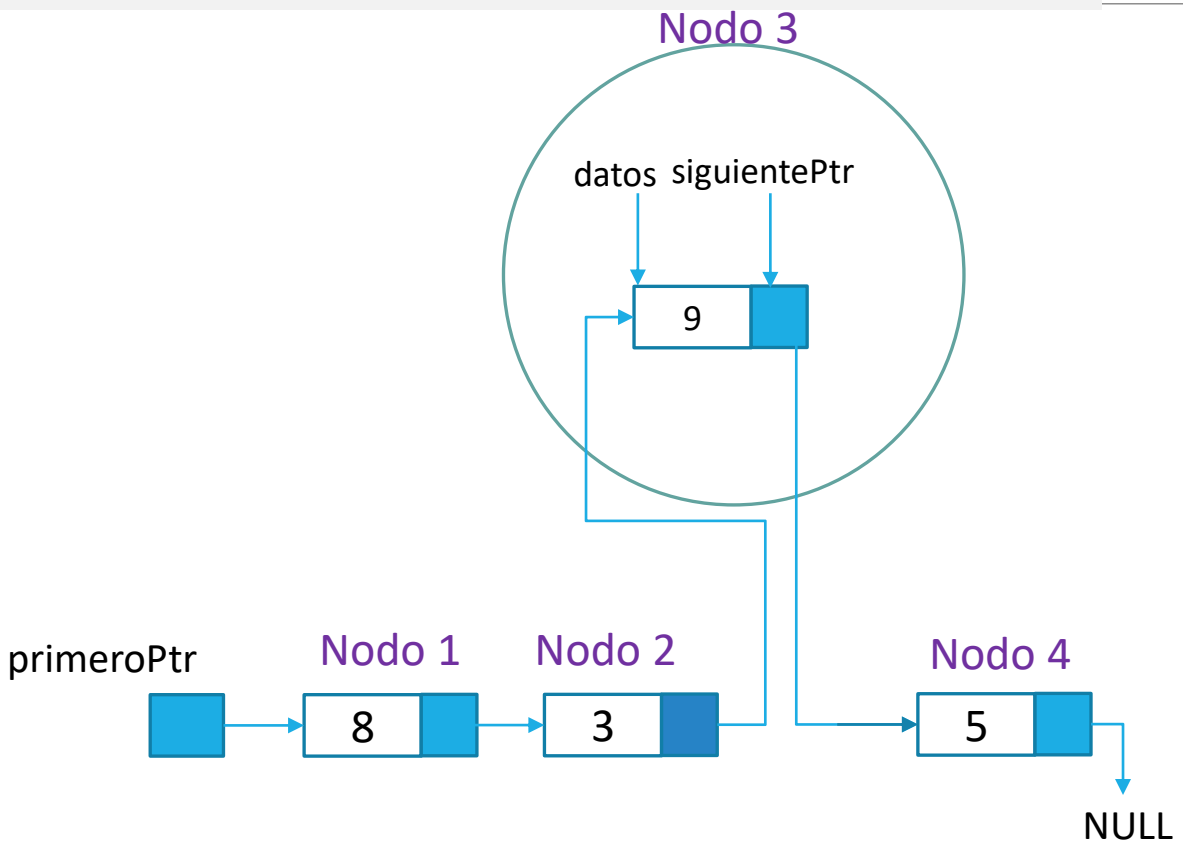


Listas ligadas o enlazadas.

- Insertar Antes

```
void Lista::insertarAntes(int referencia, int valor)
```

referencia = **5**, valor = **9**



Listas ligadas o enlazadas.

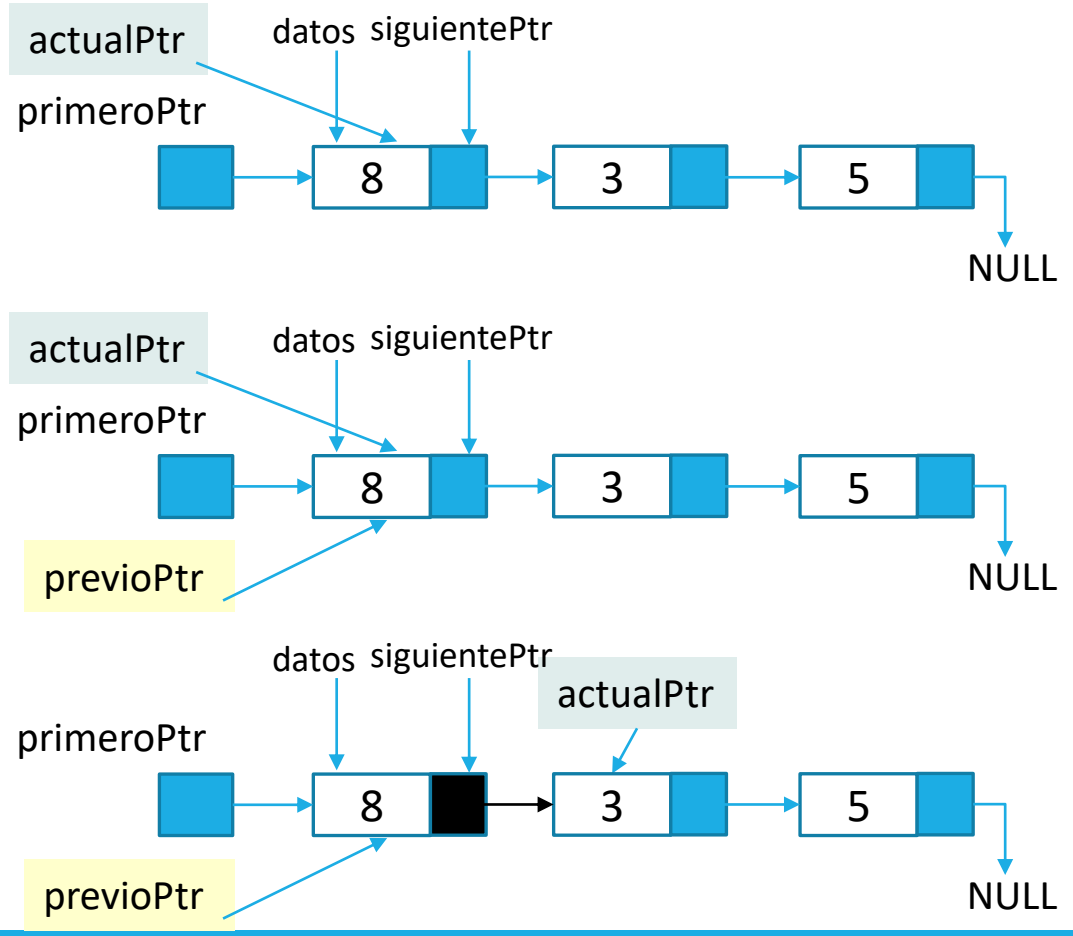
- Insertar Antes

```
void Lista::insertarAntes(int referencia, int valor)
```

referencia = **5**, valor = **9**

```
while( actualPtr->datos != referencia && BAND == 1 )
{
    if( actualPtr->siguientePtr != NULL )
    {
        previoPtr = actualPtr;  // guarda el nodo act
        actualPtr = actualPtr->siguientePtr;
    }
    else    // se llegó al final y no se encontro
    {
        BAND = 0;
    }
}
```

Buscar valor de referencia



Listas ligadas o enlazadas.

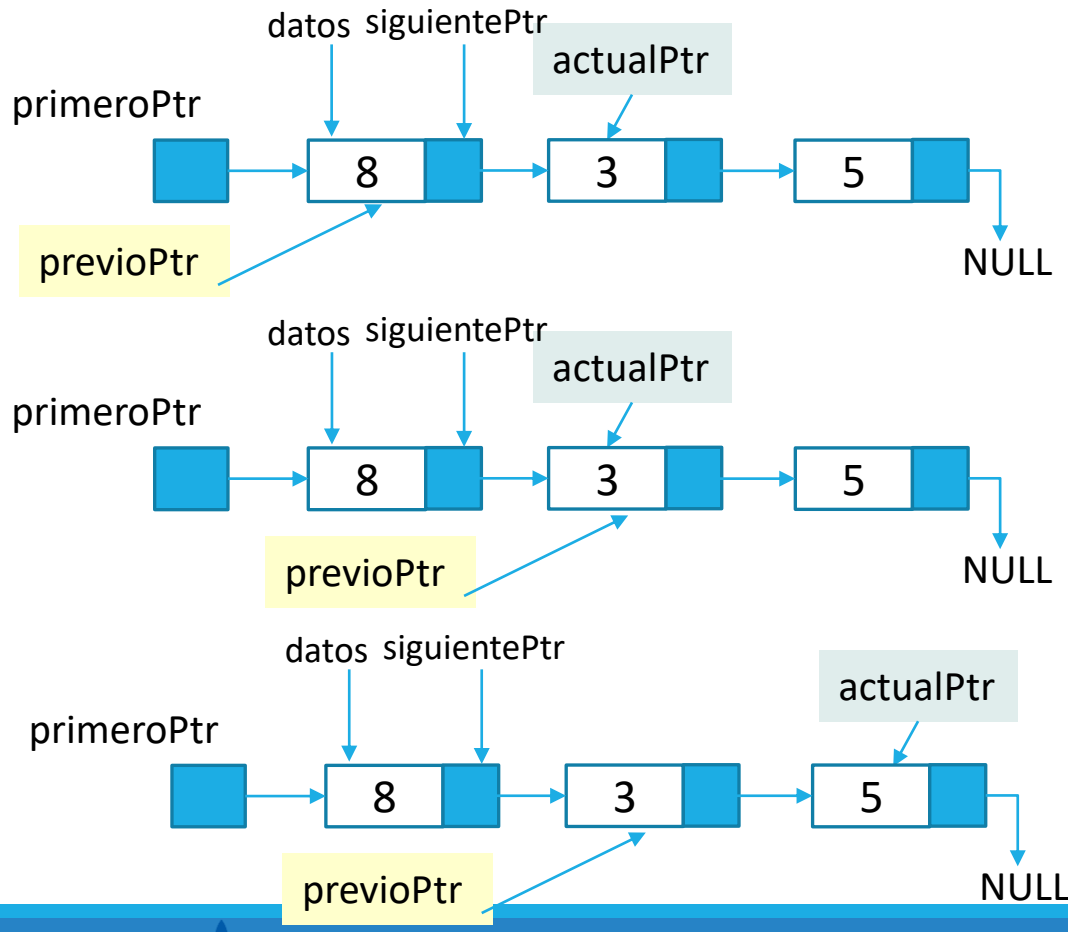
- Insertar Antes

```
void Lista::insertarAntes(int referencia, int valor)
```

referencia = **5**, valor = **9**

```
while( actualPtr->datos != referencia && BAND == 1 )
{
    if( actualPtr->siguientePtr != NULL )
    {
        previoPtr = actualPtr;  // guarda el nodo act
        actualPtr = actualPtr->siguientePtr;
    }
    else    // se llegó al final y no se encontro
    {
        BAND = 0;
    }
}
```

Buscar valor de referencia



Listas ligadas o enlazadas.

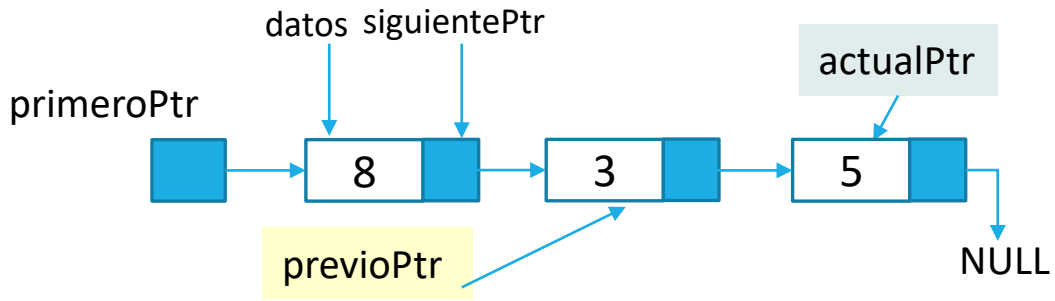
- Insertar Antes

```
void Lista::insertarAntes(int referencia, int valor)
```

referencia = **5**, valor = **9**

```
while( actualPtr->datos != referencia && BAND == 1 )
{
    if( actualPtr->siguientePtr != NULL )
    {
        previoPtr = actualPtr;  // guarda el nodo act
        actualPtr = actualPtr->siguientePtr;
    }
    else    // se llegó al final y no se encontro
    {
        BAND = 0;
    }
}
```

Buscar valor de referencia



Listas ligadas o enlazadas.

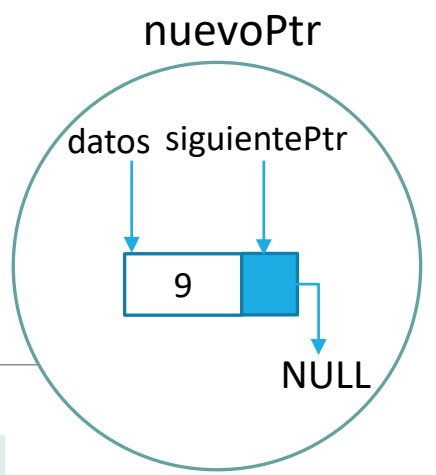
- Insertar Antes

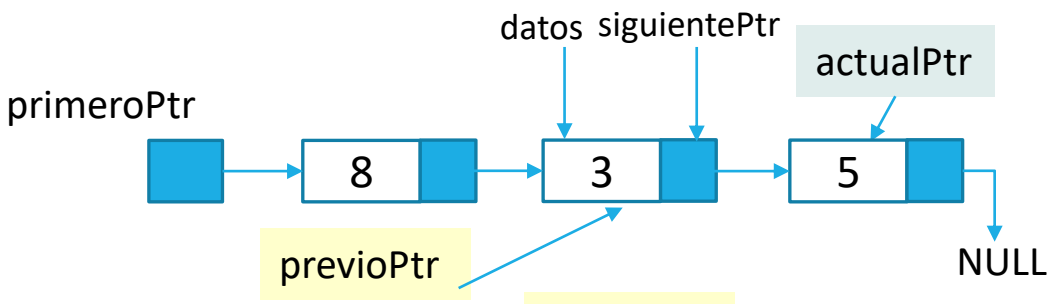
```
void Lista::insertarAntes(int referencia, int valor)
```

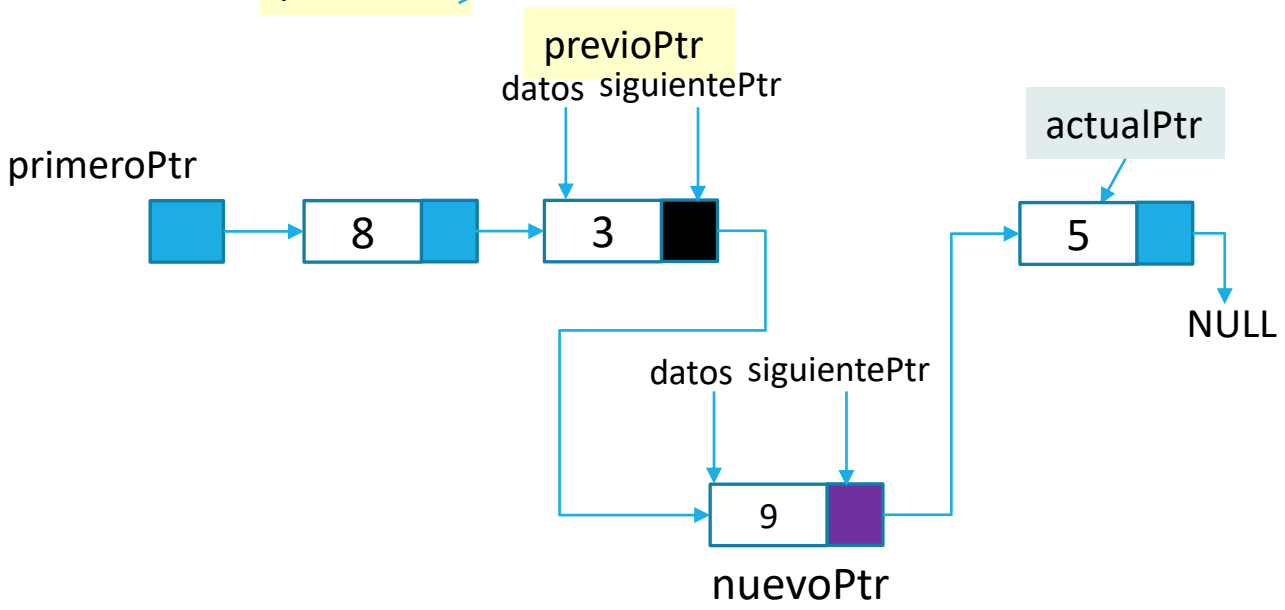
referencia = **5**, valor = **9**

```
if( BAND == 1 ) // si se encontro el nodo de
{
    if( primeroPtr == actualPtr )   // si el
    {
        nuevoPtr->siguientePtr = primeroPtr;
        primeroPtr = nuevoPtr;       // el nu
    }
    else
    {

    }
}
```

Inserta el nuevo nodo nuevoPtr con datos=9







Listas ligadas o enlazadas.

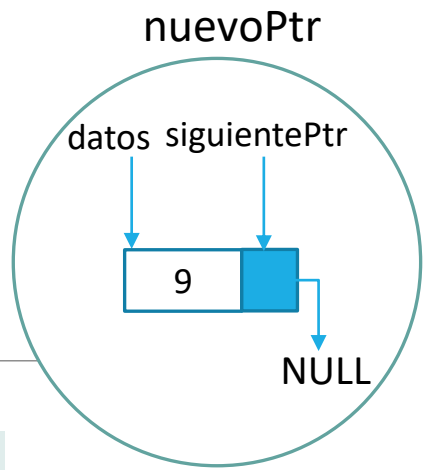
- Insertar Antes

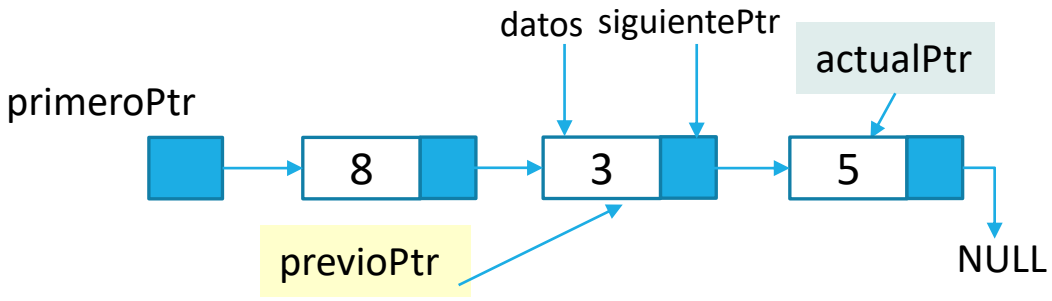
```
void Lista::insertarAntes(int referencia, int valor)
```

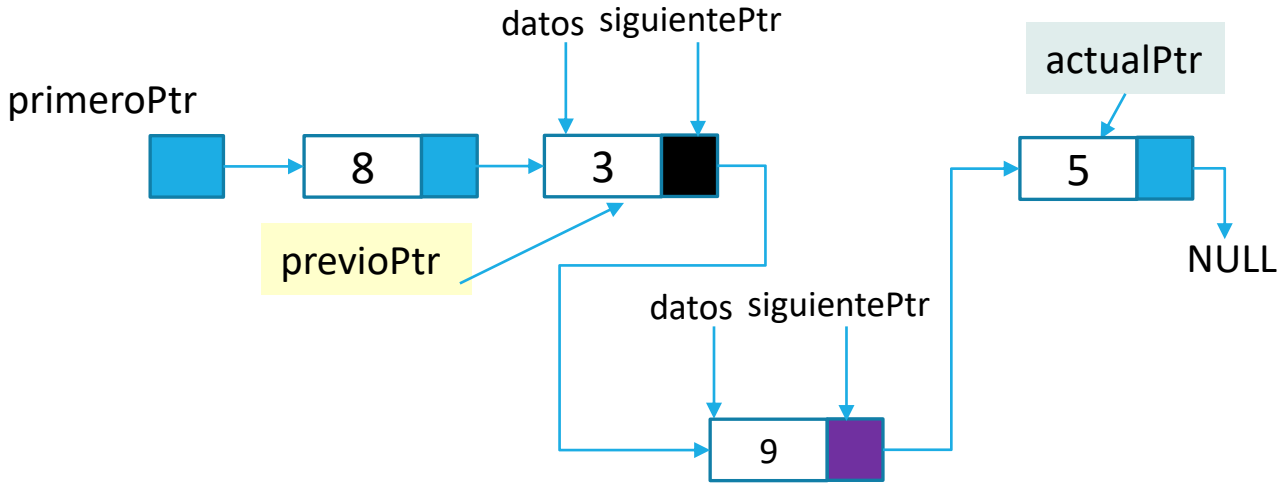
referencia = **5**, valor = **9**

```
if( BAND == 1 ) // si se encontro el nodo de
{
    if( primeroPtr == actualPtr )   // si el
    {
        nuevoPtr->siguientePtr = primeroPtr;
        primeroPtr = nuevoPtr;       // el nu
    }
    else
    {
        previoPtr->siguientePtr = nuevoPtr;
        nuevoPtr->siguientePtr = actualPtr;
    }
}
```

Inserta el nuevo nodo nuevoPtr con datos=9







Listas ligadas o enlazadas.

- **Actividad**:

De acuerdo a las operaciones estudiadas con listas simplemente enlazadas, elabora el método:

void Lista::insertarDespues(int referencia, int valor)

Es decir, el método para insertar un nodo nuevo con un valor dado, después del nodo que contenga un valor de referencia determinado.

Agrega la operación al menú principal del programa.

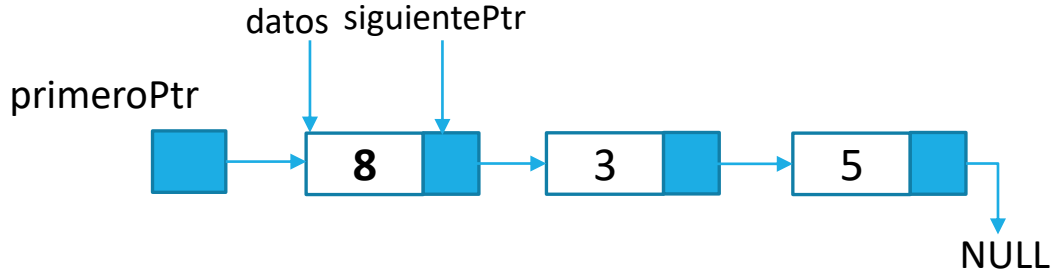
Listas ligadas o enlazadas.
Operaciones Complementarias.

- Eliminar al Inicio
- Eliminar Nodo

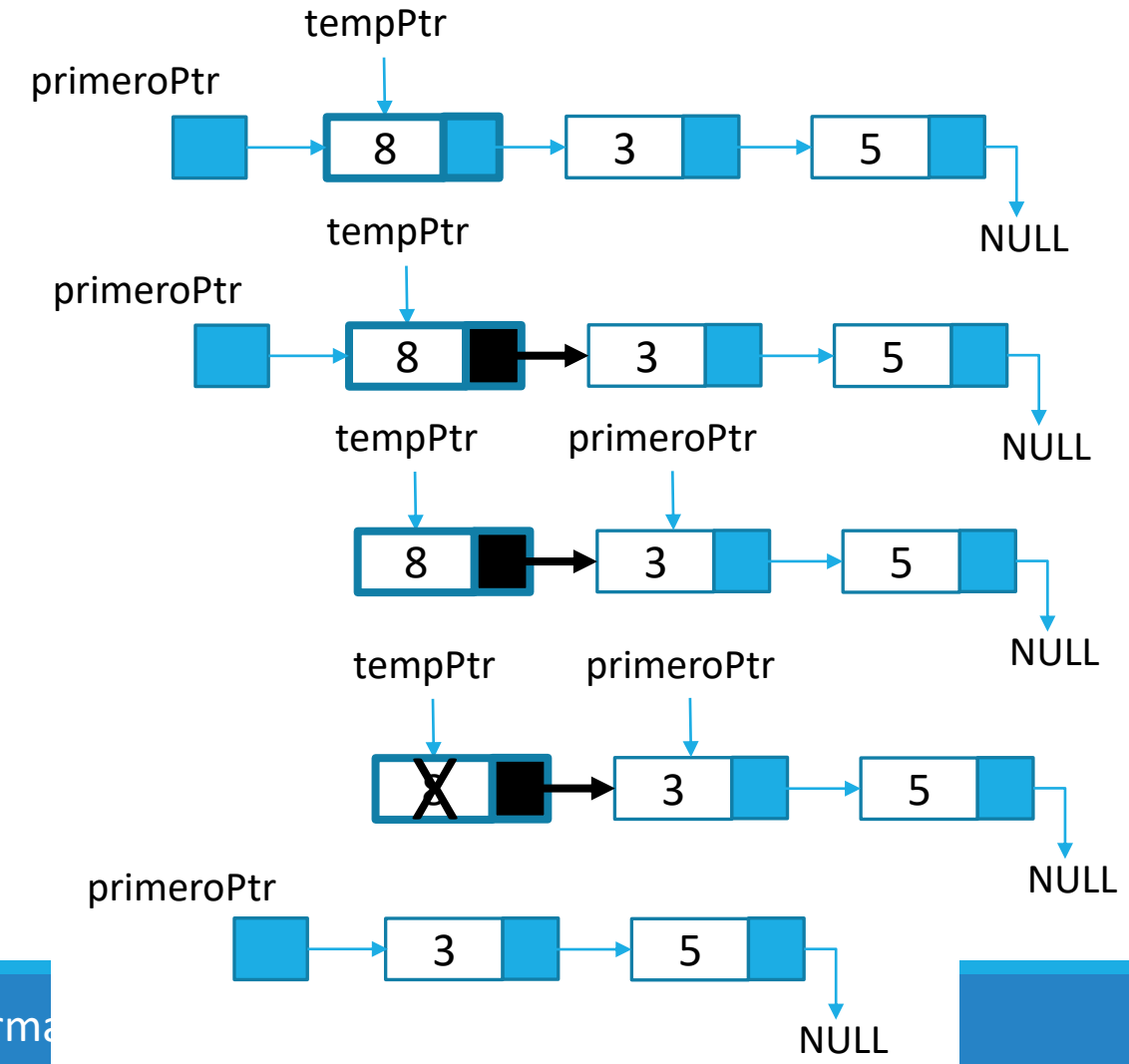
Listas ligadas o enlazadas.

- Eliminar al inicio

```
void Lista::eliminarPrimero()
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\nLa lista esta vacia\n\n";
    }

    cout << "\nDestruyendo el nodo: " << primeroPtr->datos << "\n";
    Nodo *tempPtr = primeroPtr;              // guardo el nodo elimin
    primeroPtr = primeroPtr->siguientePtr;   // el nodo segundo se co
    delete tempPtr;                          // libero la memoria del
}
```



Escribe en pantalla: Destruyendo el nodo: 8



Listas ligadas o enlazadas.

- Eliminar nodo referencia

```
void Lista::eliminarNodo(int referencia)
```

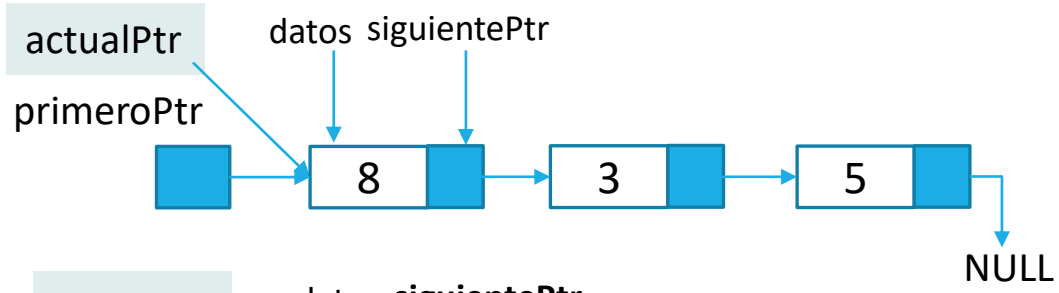
```
void Lista::eliminarNodo(int referencia)
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\nEl nodo dado como referencia no se encuentra en la lista.\n";
        return;
    }

    Nodo *actualPtr = primeroPtr;   // obtiene el primer nodo de la lista
    Nodo *previoPtr;                // variable que guarda el nodo anterior al actu

    // recorre hasta encontrar el valor o llegar al último nodo
    while( actualPtr->datos != referencia && actualPtr->siguientePtr != NULL )
    {
        previoPtr = actualPtr;  // guarda el nodo actual antes de saltar al siguien
        actualPtr = actualPtr->siguientePtr;
    }

    if( actualPtr->datos == referencia )    // si se encontro el nodo de referencia
    {
        Nodo *tempPtr = actualPtr;  // guardo el nodo eliminar

        if( primeroPtr == actualPtr )   // si el nodo de referencia es el primero
        {
            primeroPtr = primeroPtr->siguientePtr;  // el primero apunta al siguien
        }
        else
        {
            previoPtr->siguientePtr = actualPtr->siguientePtr;  // el nodo previo,
        }

        cout << "\nDestruyendo el nodo: " << tempPtr->datos << "\n";
        delete tempPtr; // libero la memoria del ultimo nodo
    }
    else
    {
        cout << "\nEl nodo dado como referencia no se encuentra en la lista.\n";
    }
}
```

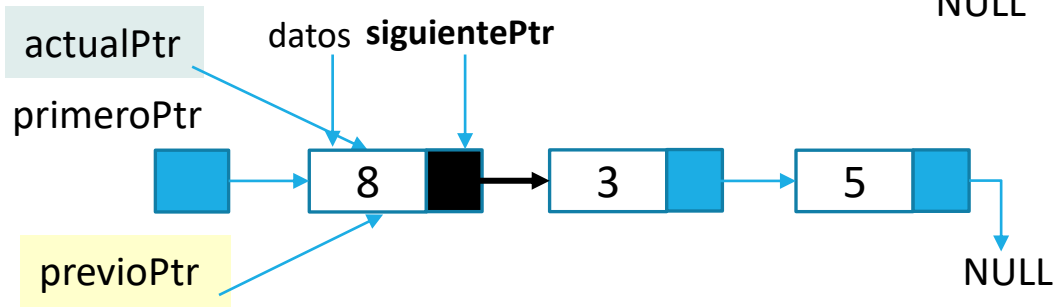
Eliminar nodo

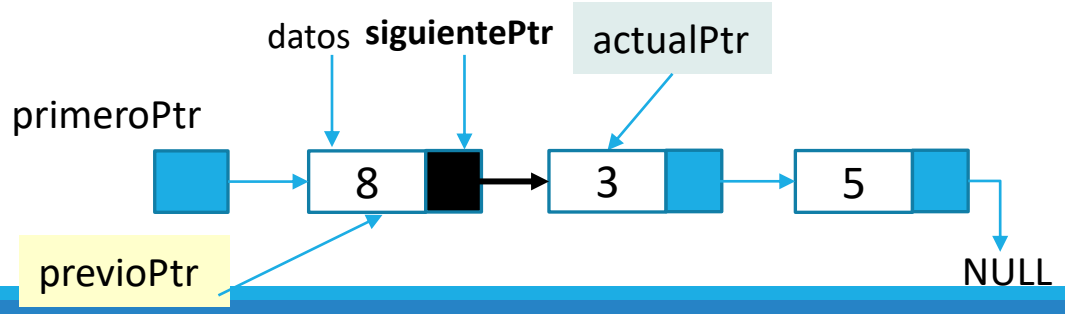
```
void Lista::eliminarNodo(int referencia)
```

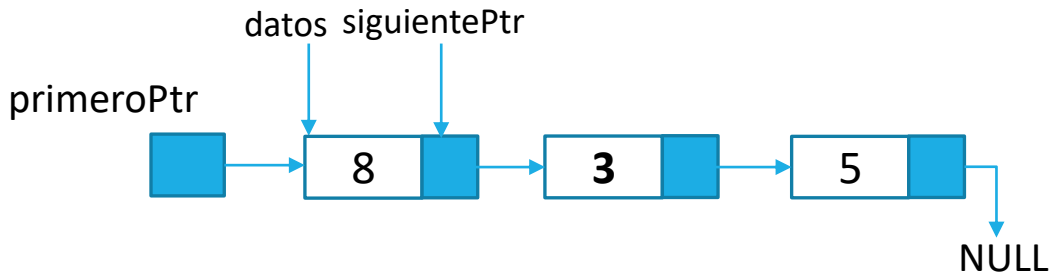
referencia=**3**

```
Nodo *actualPtr = primeroPtr;    // obtiene el primer nodo de la lista
Nodo *previoPtr;                 // variable que guarda el nodo anterior al actual

// recorre hasta encontrar el valor o llegar al último nodo
while( actualPtr->datos != referencia && actualPtr->siguientePtr != NULL )
{
    previoPtr = actualPtr;  // guarda el nodo actual antes de saltar al siguiente
    actualPtr = actualPtr->siguientePtr;
}
```





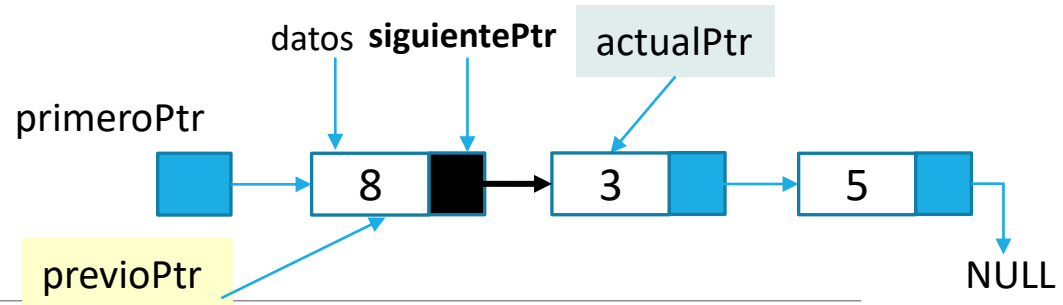


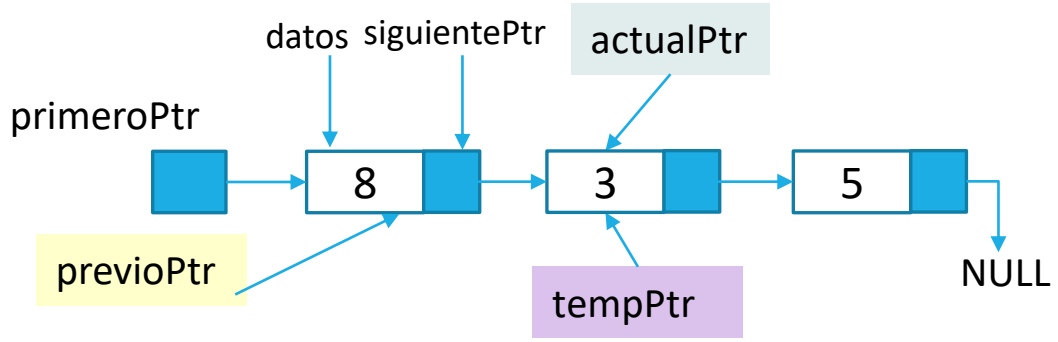


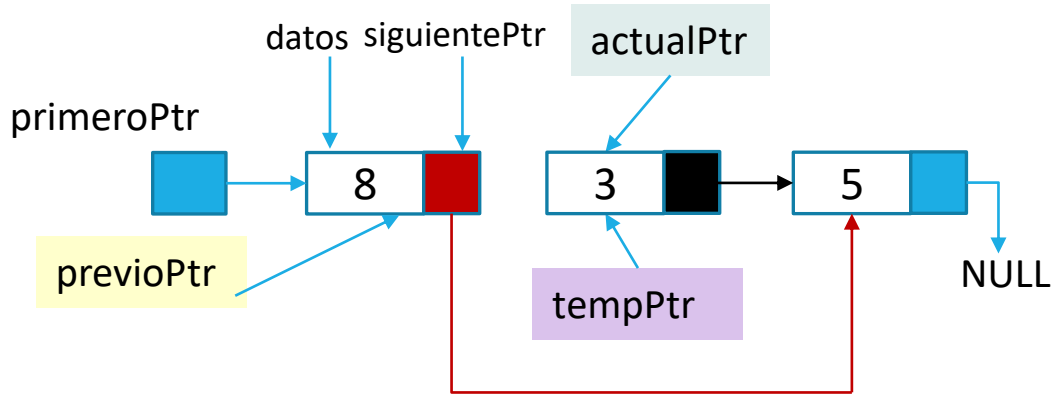
Listas ligadas o enlazadas.

- Eliminar nodo referencia

```
void Lista::eliminarNodo(int referencia) referencia=3
if( actualPtr->datos == referencia )    // si se encontro el nod
{
    Nodo *tempPtr = actualPtr;  // guardo el nodo eliminar

    if( primeroPtr == actualPtr )   // si el nodo de referencia
    {
        primeroPtr = primeroPtr->siguientePtr;  // el primero ap
    }
    else
    {
        previoPtr->siguientePtr = actualPtr->siguientePtr;  // e
    }

    cout << "\nDestruyendo el nodo: " << tempPtr->datos << "\n";
    delete tempPtr; // libero la memoria del ultimo nodo
}
else
{
    cout << "\nEl nodo dado como referencia no se encuentra en la lista.\n";
}
```





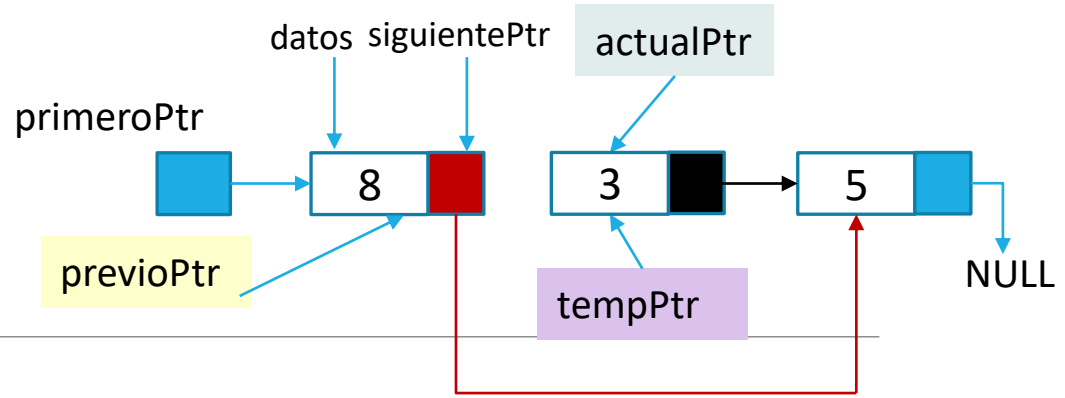


Listas ligadas o enlazadas.

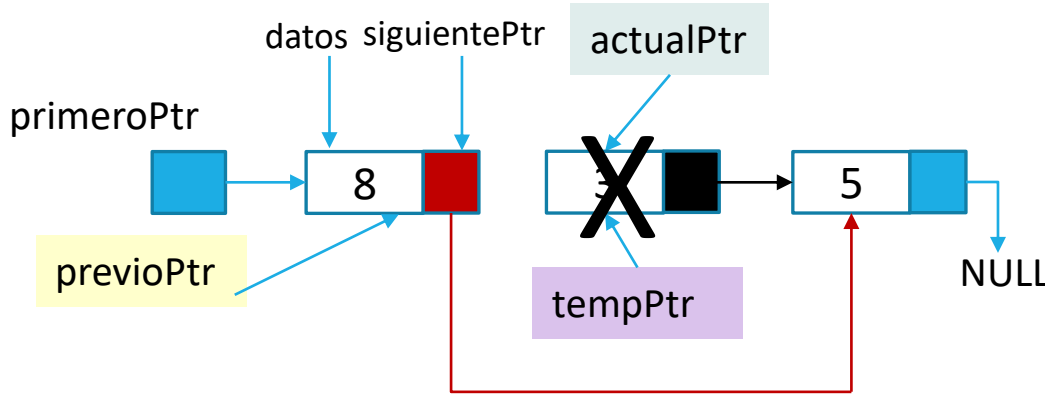
- Eliminar nodo referencia

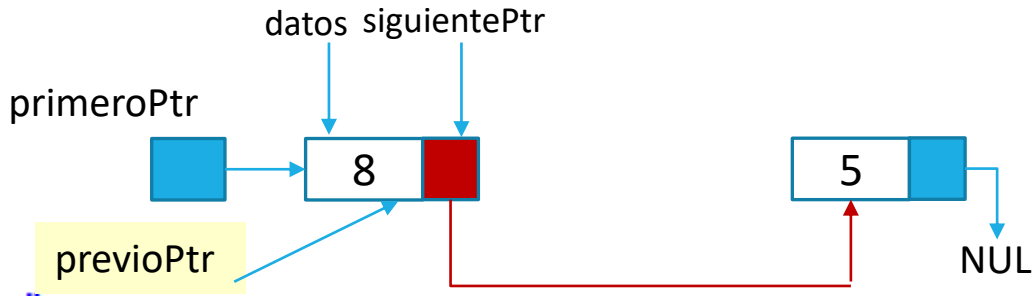
```
void Lista::eliminarNodo(int referencia)
if( actualPtr->datos == referencia )    // si se encontro el nod
{
    Nodo *tempPtr = actualPtr;  // guardo el nodo eliminar

    if( primeroPtr == actualPtr )   // si el nodo de referencia
    {
        primeroPtr = primeroPtr->siguientePtr;  // el primero ap
    }
    else
    {
        previoPtr->siguientePtr = actualPtr->siguientePtr;  // e
    }

    cout << "\nDestruyendo el nodo: " << tempPtr->datos << "\n";
    delete tempPtr; // libero la memoria del ultimo nodo
}
else
{
    cout << "\nEl nodo dado como referencia no se encuentra en la lista.\n";
}
```

referencia=**3**



Escribe en pantalla: Destruyendo el nodo: 3





Listas ligadas o enlazadas.

- **Actividad**:

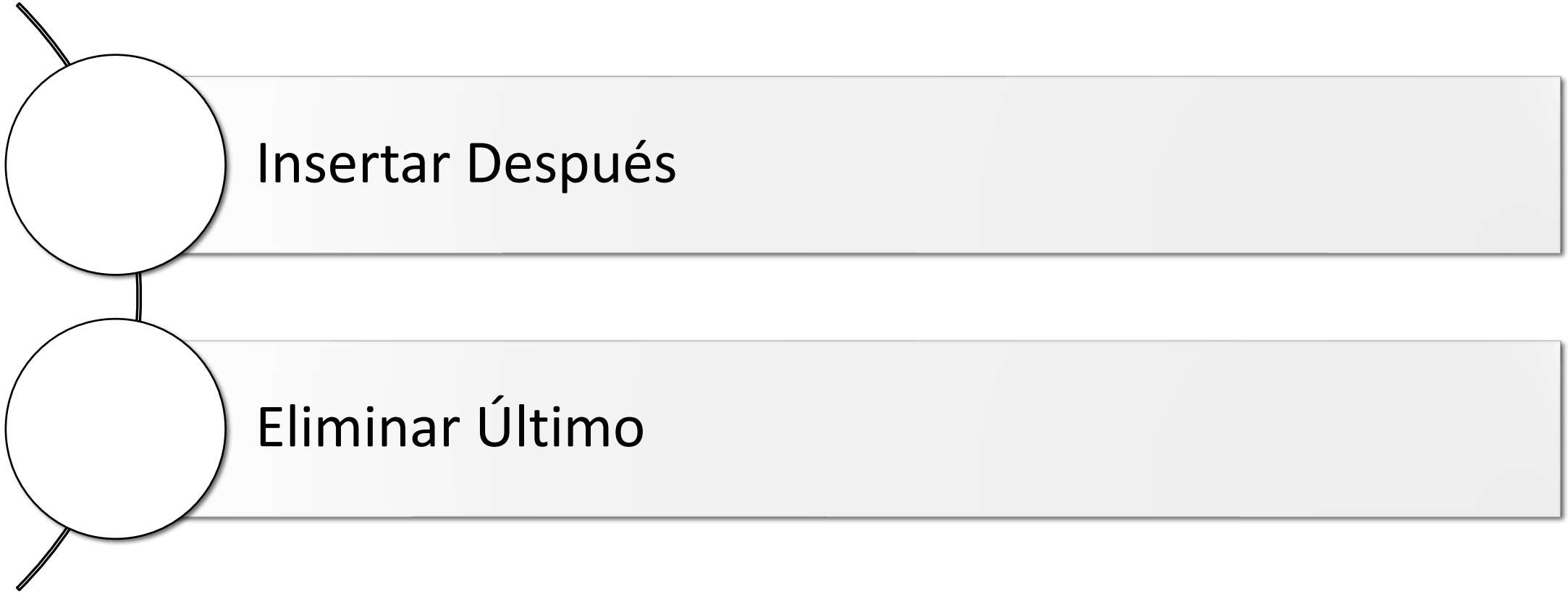
De acuerdo a las operaciones estudiadas con listas simplemente enlazadas, elabora el método:

void Lista::eliminarUltimo

Es decir, el método para eliminar el último nodo de la lista.

Agrega la operación al menú principal del programa.

Listas ligadas o enlazadas.
Operaciones Complementarias.



Listas ligadas o enlazadas.

- Insertar Después

```
void Lista::insertarDespues(int referencia, int valor)
{
    if( estaVacia() )   // la lista esta vacia
    {
        cout << "\nEl nodo dado como referencia no se encuentra en la lista. ";
        system("pause");
        return;
    }

    Nodo *nuevoPtr = new Nodo();     // Q, variable nodo temporal
    nuevoPtr->datos = valor;         // DATO, se guarda el nuevo valor
    nuevoPtr->siguientePtr = NULL;   // el nuevo nodo apunta a nulo

    Nodo *actualPtr = primeroPtr;    // obtiene el primer nodo de la lista

    // recorre hasta encontrar el valor o llegar al último nodo
    while( actualPtr->datos != referencia && actualPtr->siguientePtr != NULL )
    {
        actualPtr = actualPtr->siguientePtr;
    }

    if( actualPtr->datos == referencia )    // si se encontro el nodo de refere
    {
        nuevoPtr->siguientePtr = actualPtr->siguientePtr;   // el nodo nuevo, ap
        actualPtr->siguientePtr = nuevoPtr;                 // el nodo referenc
    }
    else
    {
        cout << "\nEl nodo dado como referencia no se encuentra en la lista. ";
    }
}
```

Listas ligadas o enlazadas.

- Eliminar Último

```
void Lista::eliminarUltimo()
{
    if( estaVacia() )   // la lista esta vacia
    {
        return;
    }

    Nodo *actualPtr = primeroPtr;    // obtiene el primer nodo de
    Nodo *previoPtr;                 // variable que guarda el nod

    while( actualPtr->siguientePtr != NULL )    // recorre hasta
    {
        previoPtr = actualPtr;  // guarda el nodo actual antes de
        actualPtr = actualPtr->siguientePtr;
    }

    Nodo *tempPtr = actualPtr;  // guardo el nodo eliminar

    if( primeroPtr == actualPtr )   // solo hay un nodo
    {
        primeroPtr = NULL;  // se vacía la lista
    }
    else
    {
        previoPtr->siguientePtr = NULL; // el nodo previo, se vue
    }

    cout << "\nDestruyendo el nodo: " << tempPtr->datos << "\n";
    delete tempPtr; // libero la memoria del ultimo nodo
}
```

Listas ligadas o enlazadas.

Aplicaciones de las listas

Dos aplicaciones relevantes de las listas son:

- Representación de polinomios.
- Resolución de Colisiones (*hash*).

En general, las listas son muy útiles para las aplicaciones en las que se requiere dinamismo en el crecimiento y una estructura de datos reducida para el almacenamiento de información.

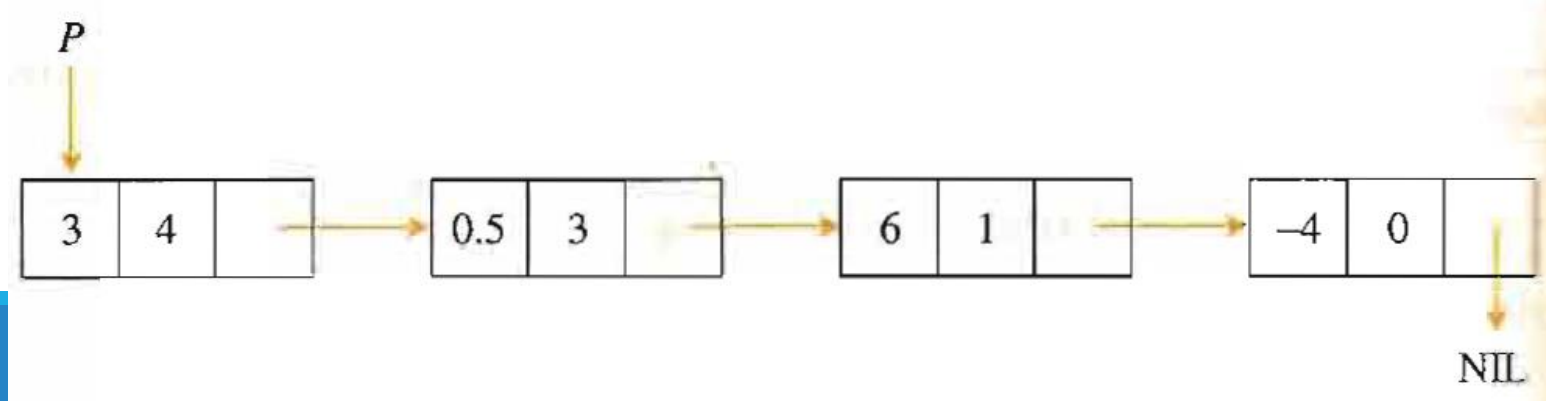
Listas ligadas o enlazadas.

Aplicaciones

- **Representación de polinomios.**
- Las listas se pueden emplear para almacenar los coeficientes diferentes de cero del polinomio, junto al exponente. Así, por ejemplo, dado el polinomio:

$$P(x) = 3X^4 + 0.5X^3 + 6X - 4$$

- Su representación mediante listas es la siguiente:

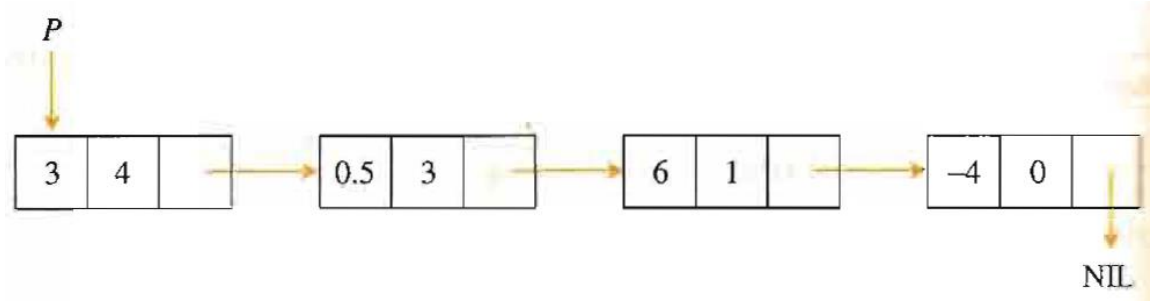


Listas ligadas o enlazadas.

Aplicaciones

- **Representación de polinomios.**
- El campo INFORMACIÓN de cada nodo de la lista contendrá dos campos: el *Coeficiente* y el *Exponente*.

$$P(x) = 3X^4 + 0.5X^3 + 6X - 4$$



Listas ligadas o enlazadas.

Aplicaciones

- **Solución de Colisiones (Hash)**
- La búsqueda es una de las operaciones más importantes en el procesamiento de información. Se utiliza básicamente para recuperar datos que se habían almacenado previamente.
- La búsqueda interna se realiza cuando todos los datos se encuentran en memora principal. Arreglos, listas, etc.
- La búsqueda externa si los elementos se encuentran en memoria secundaria.

Listas ligadas o enlazadas.

Aplicaciones

- **Solución de Colisiones (Hash)**
- Los métodos de búsqueda interna más importantes son:
  - Secuencial o lineal.
  - Binaria.
  - Por transformación de claves o hash.
  - Árboles de búsqueda.

Listas ligadas o enlazadas.

Aplicaciones

- **Solución de Colisiones (Hash)**
- Por transformación de claves o hash.
  - Permite aumentar la velocidad de búsqueda sin necesidad de tener los elementos ordenados.
  - El tiempo de búsqueda es independiente del número de elementos del arreglo.
  - Permite localizar un dato de forma directa, es decir, sin tener que recorrer algunos elementos antes de encontrarlo.
  - Utiliza una función que convierte una clave dada en una dirección, índice, dentro del arreglo.

Listas ligadas o enlazadas.

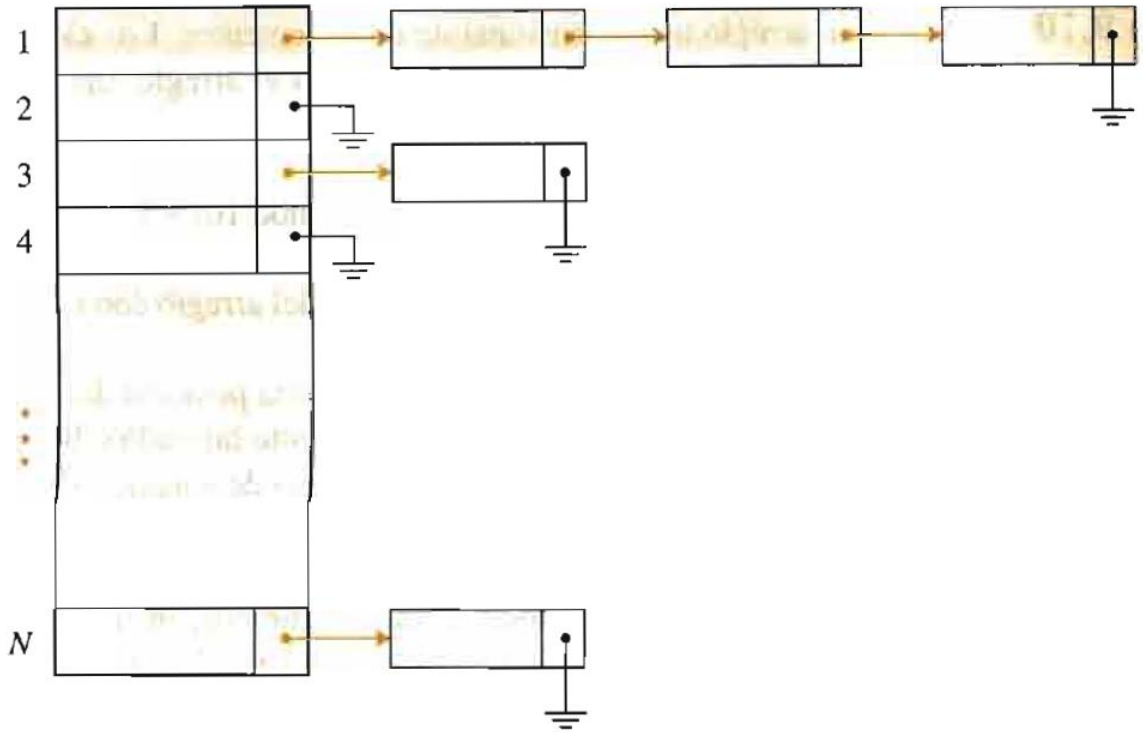
Aplicaciones

- **Solución de Colisiones (Hash)**
- Por transformación de claves o hash.
  - La función Hash aplicada a la clave genera un índice del arreglo que permite tener acceso de manera directa al elemento.
  - Esta función Hash debe ser simple de calcular y asignar direcciones de manera uniforme, es decir, asignar posiciones distintas a dos claves distintas.
  - Si ocurre la asignación de una misma dirección a dos o mas claves distintas, se genera una colisión.

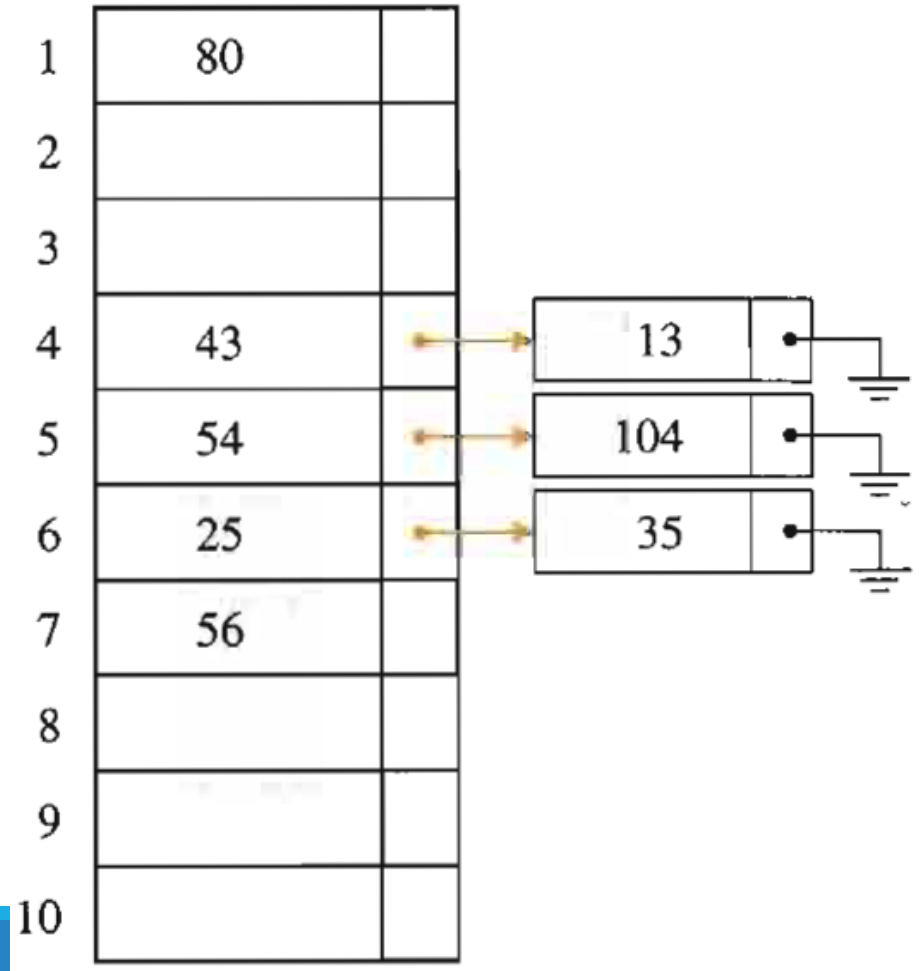
Listas ligadas o enlazadas.

Aplicaciones

- **Solución de Colisiones (Hash)**
- Solución de colisiones por método de **Encadenamiento**
  - Consiste en que cada elemento del arreglo tenga un apuntador a una lista ligada, la cual se irá generando y almacenará los valores que colisionan.



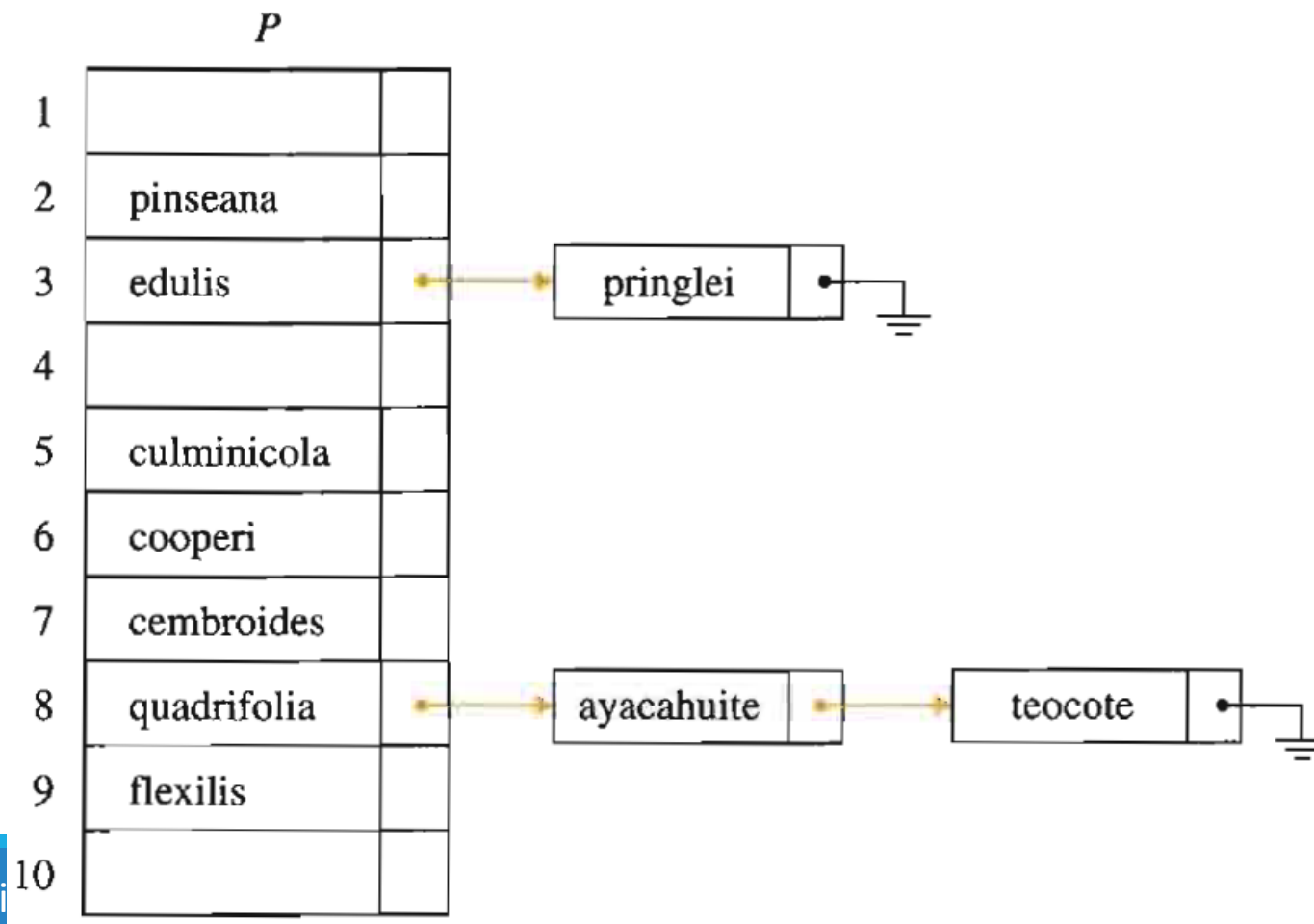
Solución de Colisiones por encadenamiento



K	H(K)
25	6
43	4
56	7
35	6
54	5
13	4
80	1
104	5

Solución de Colisiones por encadenamiento

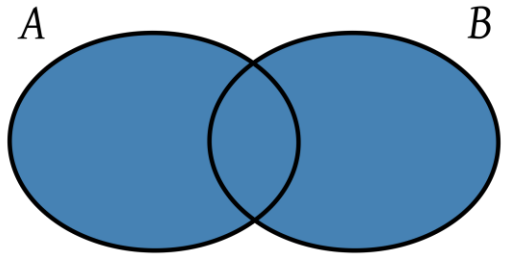
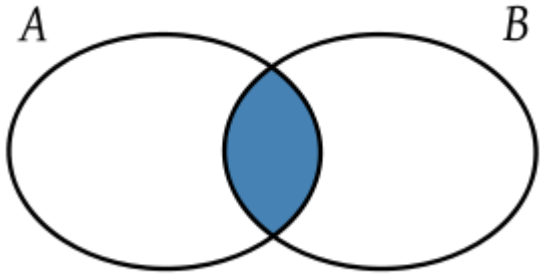
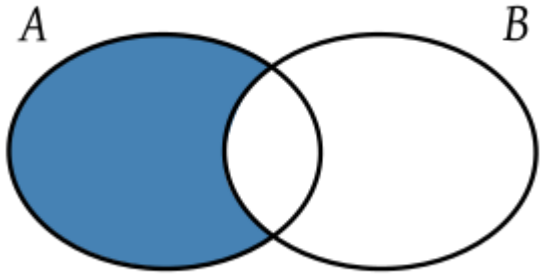
Nombre	Valor numérico	Dirección
Cembroides	96	7
Edulis	72	3
Culminicola	114	5
Quadrifolia	117	8
Pinseana	81	2
Flexilis	98	9
Ayacahuite	97	8
Teocote	87	8
Cooperi	85	6
Pringlei	92	3



Listas ligadas o enlazadas.

Actividad:

Implementar las operaciones de unión, intersección y diferencia de conjuntos haciendo uso de listas simplemente ligadas.

Unión: {1,2,3} ∪ {2,4,6} = {1,2,3,4,6}	**Intersección**: {1,2,3} ∩ {2,4,6} = {2}	**Diferencia**: {1,2,3} - {2,4,6} = {1,3}
		

Bibliografía

Cairó, O. y Guardati, S. (2002). Estructuras de Datos, 2da. Edición. McGraw-Hill.

Deitel P.J. y Deitel H.M. (2008) Cómo programar en C++. 6ª edición.Prentice Hall.

Joyanes, L. (2006). Programación en C++: Algoritmos, Estructuras de datos y objetos. McGraw-Hill.