

Algoritmos y

Estructuras

De

Datos

ESTRUCTURAS DE DATOS

DINAMICAS

Estructuras de datos dinámicas

Estructuras de datos que crecen y se achican durante la ejecución de un programa.

Son estructuras de datos dinámicas:

- **Arrays dinámicos**

- Se crean en tiempo de ejecución, a diferencia de los arrays estáticos que se crean en tiempo de compilación.
- Se les asigna memoria recién al ejecutarse su declaración.

- **Listas enlazadas**

- Colección de elementos dispuestos uno a continuación del otro.
- Permiten inserciones y borrados en cualquier parte de la lista.

- **Pilas**

- Permiten inserciones y borrados sólo en el tope (parte superior) de la pila.

- **Colas**

- Permiten inserciones desde un extremo y borrados en el otro. Específicamente, las inserciones se hacen en su parte final, y las eliminaciones en su parte inicial.

ESTRUCTURAS DE DATOS: Clasificaciones

- Según donde se almacenan {
 - Internas** (en memoria principal)
 - Externas** (en memoria auxiliar)
- Según los tipos de datos de sus componentes {
 - Homogéneas** (todas del mismo tipo)
 - No homogéneas** (pueden ser de distinto tipo)
- Según la implementación {
 - Provistas por los lenguajes** (básicas)
 - Abstractas** (TDA - Tipo de dato abstracto que puede implementarse de diferentes formas)
- Según la forma de almacenamiento {
 - Estáticas** (ocupan posiciones fijas y su tamaño nunca varía durante todo el módulo)
 - Dinámicas** (su tamaño varía durante el módulo y sus posiciones también)

Asignación Dinámica de Memoria

Posibilita hacer un uso más eficiente de la memoria disponible, evitando que las variables y estructuras “ocupen espacio” cuando no se usan.

El espacio de las variables asignadas dinámicamente se crea durante la ejecución del programa, y no como en el caso de variables estáticas cuyo espacio se asigna en tiempo de compilación.

El límite para dicha asignación puede ser tan grande como la cantidad de memoria física disponible en la computadora.

La **asignación** se realiza con la instrucción **new** y el **tipo de dato** para el que se quiere reservar memoria.

Ejemplo:

```
Ptr = new int;
```

 Tipo de la variable dinámica a crear

Reserva Dinámica de Memoria

La **memoria dinámica** se obtiene y libera durante la **ejecución** del programa.

La memoria dinámica la debe gestionar el programador. Cuando necesite crear una variable dinámica deberá solicitar memoria dinámica con el operador:

■ **new**

- Solicita reserva de memoria de un número de bytes.
- Retorna un puntero de tipo `void *`
 - Es un puntero a la dirección del bloque asignado de memoria.
 - Un puntero `void *` puede ser asignado a cualquier tipo de datos.
- Si no hay memoria disponible, retorna NULL.

La **asignación dinámica de memoria** es muy simple: una única expresión realiza todo el trabajo de cálculo de tamaño, asignación, comprobaciones de seguridad y conversión de tipo.

Asignación Dinámica de Memoria

La **asignación** se realiza con la instrucción `new` y el `tipo de dato` para el que se quiere reservar memoria.

Si la asignación falla, devuelve un NULL.

Una forma de averiguar si falló la asignación de memoria:

```
a = new int;
if (a == NULL) {
    cout << "ERROR, no hay memoria suficiente";
}
else{
    .....
}
```

Otra forma, más elegante:

```
if ((a = new int) == NULL) {
    cout << "ERROR, no hay memoria suficiente";
}
else{
    .....
}
return 0;
```

Reserva Dinámica de Memoria: ejemplos

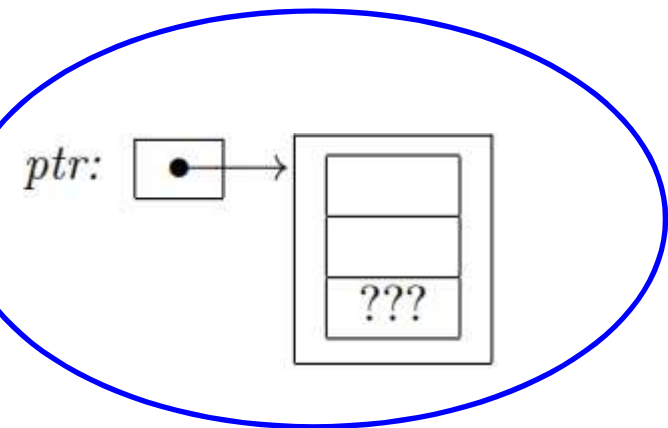
- Ejemplo 1:

```
ptr = new int;
```

- Ejemplo 2:

```
struct Persona {  
    char nombre[20], apellido[20];  
    int edad; };
```

```
int main(){  
    Persona * ptr;  
    ptr = new Persona;  
    .....  
}
```



A la variable *ptr* se le asigna el valor del puntero (una dirección de memoria) que apunta a la variable dinámica de tipo *Persona* creada por el operador **new**.

Reserva Dinámica de Memoria

Cuando el programador necesite liberar la variable dinámica, deberá utilizar el operador:

■ **delete**

- Libera memoria asignada previamente por new
- El argumento para delete es un puntero a la dirección de memoria que se quiere liberar
- Ejemplo:

delete Ptr;

delete libera un único objeto.

delete [] (con corchetes) libera un vector dinámico.

No ocurre nada si el puntero que se le pasa a delete es nulo. Por esa razón se recomienda asignar cero o NULL al puntero inmediatamente después de usar delete. Se evita así que pueda ser usado de nuevo como argumento para delete.

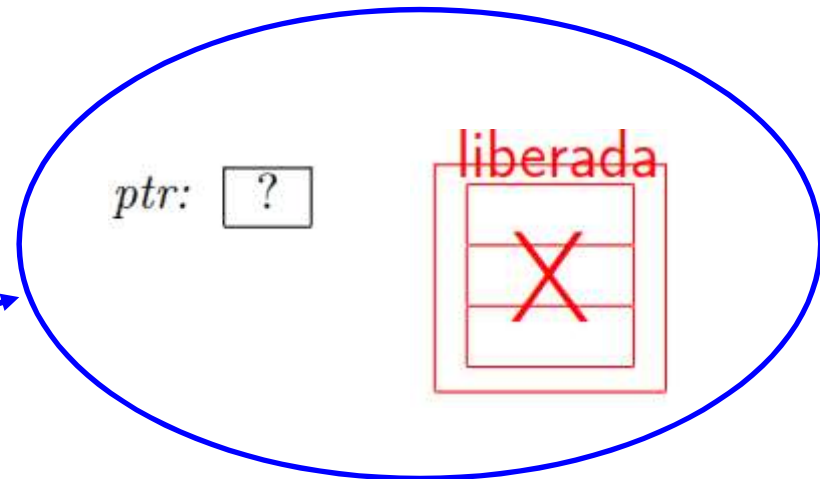
Reserva Dinámica de Memoria

- Ejemplo:

```
struct Persona {  
    char nombre[20], apellido[20];  
    int edad; };
```

```
int main(){  
    Persona * ptr;  
    ptr = new Persona;  
    .....  
    .....  
    delete ptr;  
}
```

El puntero deja de contener una dirección válida!



delete realiza dos acciones, primero destruye la variable dinámica y después libera la memoria dinámica reservada para dicha variable. Finalmente, ptr queda con un valor no especificado, y será destruida automáticamente por el compilador cuando el flujo de ejecución salga de su ámbito de declaración.

Vectores dinámicos

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      int * a;
6      int talla, i;
7
8      cout << "Numero de elementos: ";
9      cin >> talla;
10     a = new int[talla];
11     for (i=0; i < talla; i++)
12         a[i] = i;
13     delete [] a;
14     a = NULL;
15     return 0;
16 }
```

Vector a, definido como un puntero a una secuencia de enteros: *vector dinámico de enteros*

Reserva de memoria para *talla* enteros

Liberación de la memoria reservada

A un puntero que no tiene memoria reservada, se asigna que no apunte a un bloque de memoria

Si el usuario decide que *talla* valga, por ejemplo, 5, se reservará un total de 20 bytes y la memoria quedará así, luego de ejecutar la línea 10:

Es decir, se reserva memoria suficiente para alojar 5 enteros.



Estructuras Auto-Referenciadas

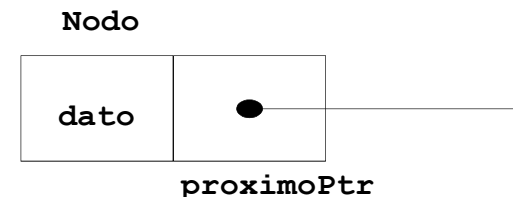
■ Estructuras auto referenciadas

- Estructuras que tienen un puntero a una estructura del mismo tipo.
- Pueden ser enlazadas para formar estructuras de datos útiles como **listas, colas, pilas y árboles**.
- Terminan con un puntero NULL (0).

```
struct Nodo {  
    int dato;  
    Nodo *proximoPtr;  
};
```

■ proximoPtr

- Apunta a un objeto de tipo nodo.
- Se conoce como un “enlace”
 - enlaza un nodo a otro nodo.



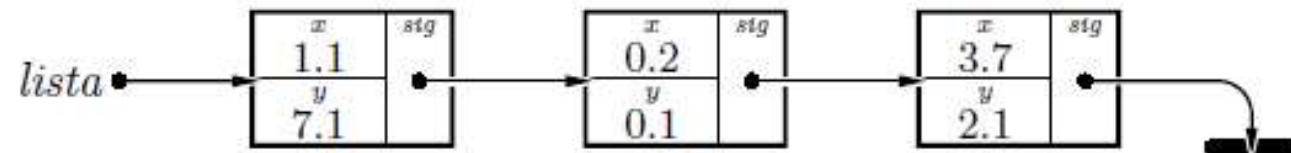
Estructuras Auto-Referenciadas

- Dos estructuras autoreferenciadas enlazadas juntas (formando una lista)



- Otro ejemplo: Una lista de puntos en el plano

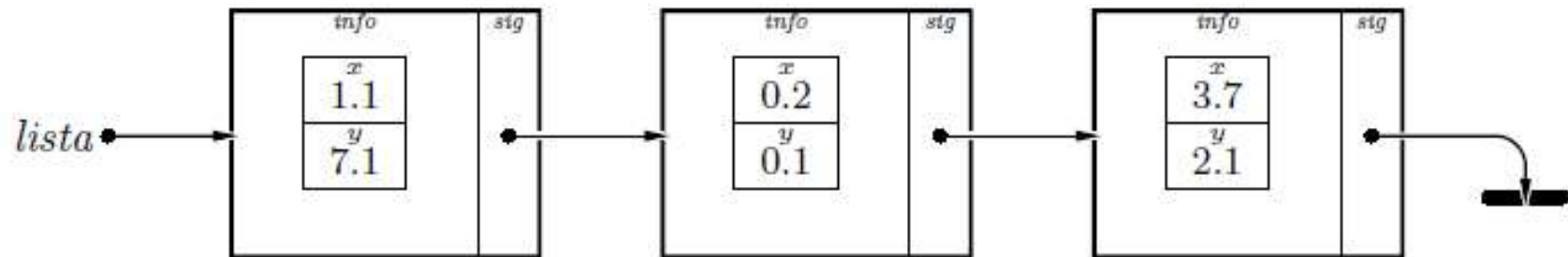
```
struct Nodo {  
    float x;  
    float y;  
    Nodo * sig;  
};
```



Estructuras Auto-Referenciadas

- **Otra forma:** se define un tipo adicional (struct Punto). Cada nodo utiliza un único campo de tipo Punto:

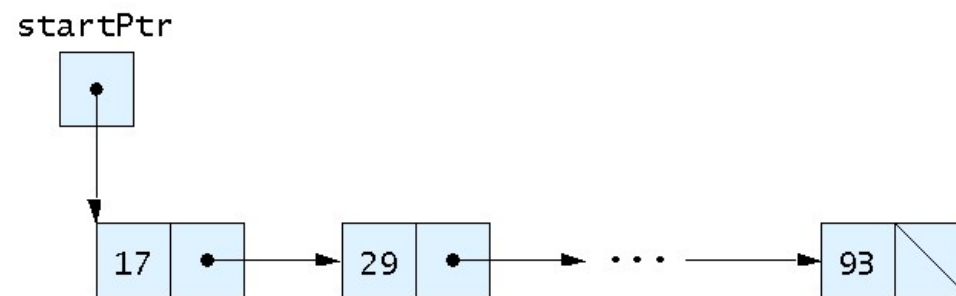
```
struct Punto {  
    float x;  
    float y;  
};  
  
struct Nodo {  
    Punto info;  
    Nodo * sig;  
};
```



Listas Enlazadas

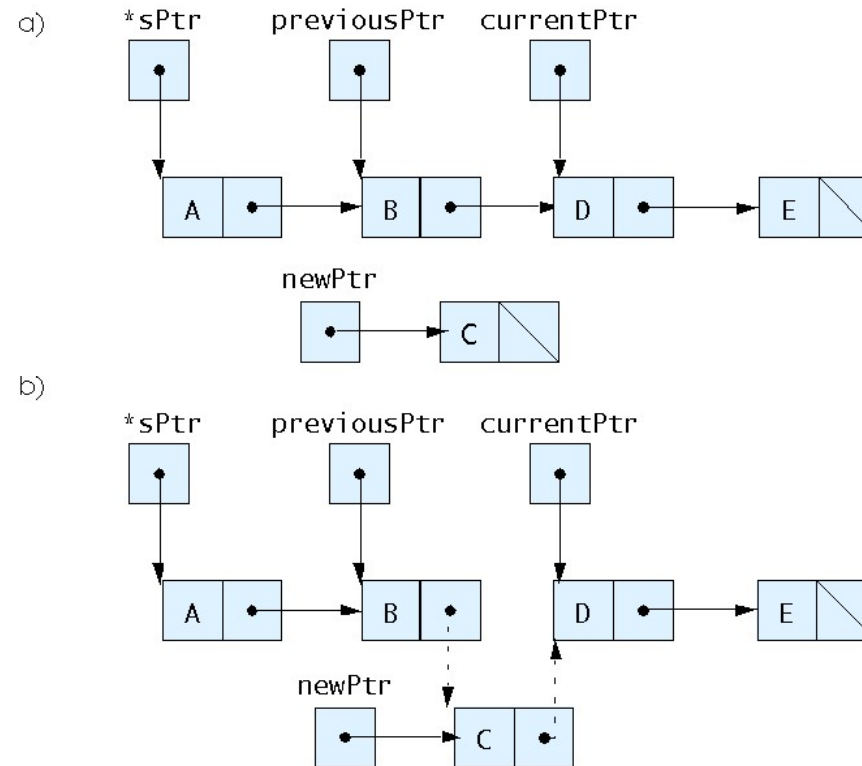
- Colección lineal de objetos autoreferenciados, llamados **nodos**.
- Conectados mediante **punteros**.
- Accedidos mediante un puntero al primer nodo de la lista (**puntero maestro**).
- Los nodos siguientes se acceden mediante el campo puntero (**enlace**) del nodo actual.
- El puntero enlace del último nodo se pone a NULL para marcar el **final de la lista**.
- Los datos se **almacenan dinámicamente**.
- Los nodos se van creando a medida que se los necesita.
- **Utilice una lista enlazada en lugar de un arreglo cuando:**
 - **No se conozca** por anticipado la cantidad de elementos de datos a representarse.
 - Su lista necesite **inserciones y eliminaciones rápidamente**.
 - La inserción y eliminación en arreglos ordenados insume mucho tiempo, ya que los elementos que se encuentran a continuación del elemento insertado/eliminado se deben desplazar adecuadamente.

Listas Enlazadas



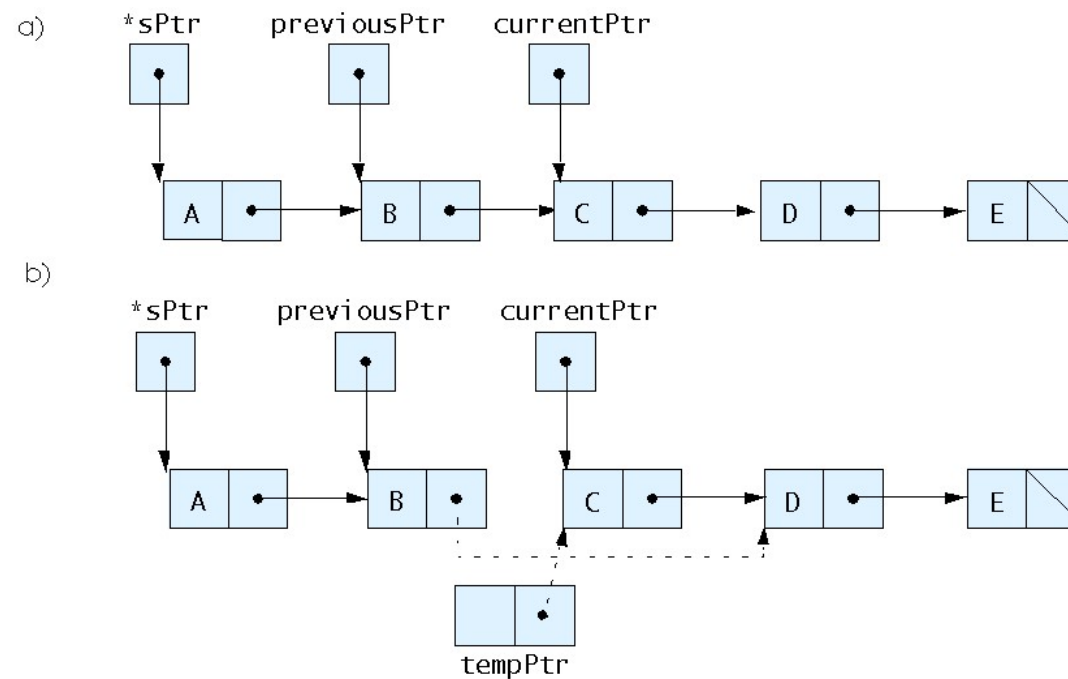
Representación gráfica de una lista enlazada

Listas Enlazadas



Inserción de un nuevo nodo

Listas Enlazadas



Eliminación de un nodo

```
struct Nodo {  
    int info;  
    Nodo * sig;  
};
```

Listas Enlazadas

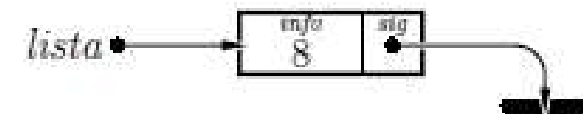
```
int main() {  
    Nodo * lista = NULL;
```

CREAR LISTA VACÍA



```
int main() {  
    Nodo * lista = NULL;  
  
    lista = new Nodo;  
    lista->info = 8;  
    lista->sig = NULL;
```

CREAR EL PRIMER ELEMENTO (NODO)

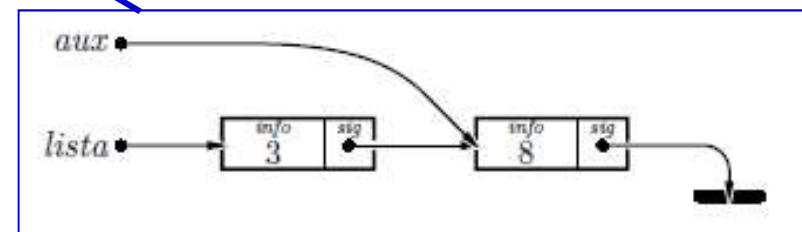
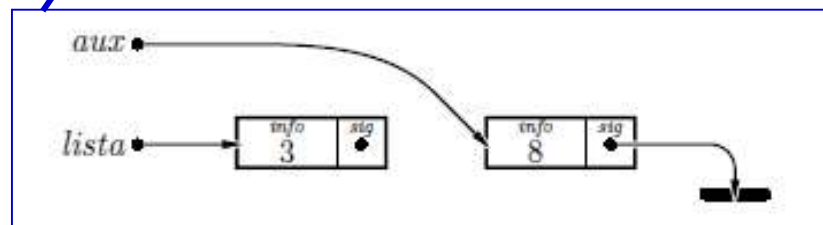
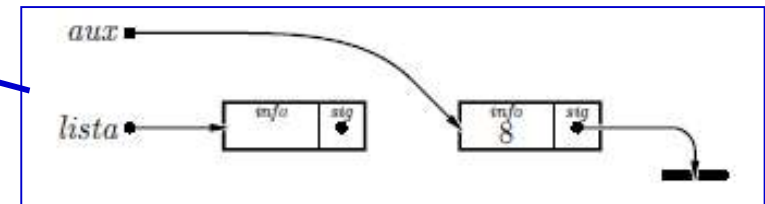
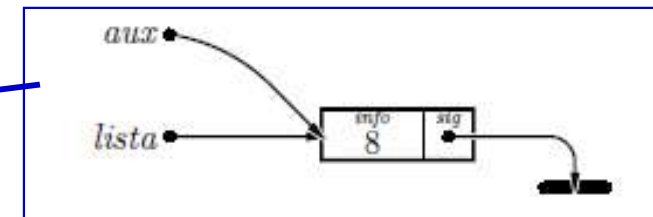


La variable *lista* es de tipo **Nodo ***, es un puntero, por eso el uso del operador \rightarrow para acceder a los campos del registro.

Listas Enlazadas

AGREGAR UN ELEMENTO AL PRINCIPIO

```
int main() {  
    Nodo * lista = NULL, * aux;  
  
    ...  
    aux = lista;  
    lista = new Nodo;  
    lista->info = 3;  
    lista -> sig = aux;  
}
```

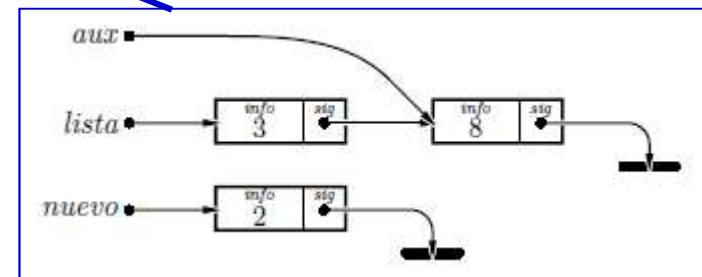
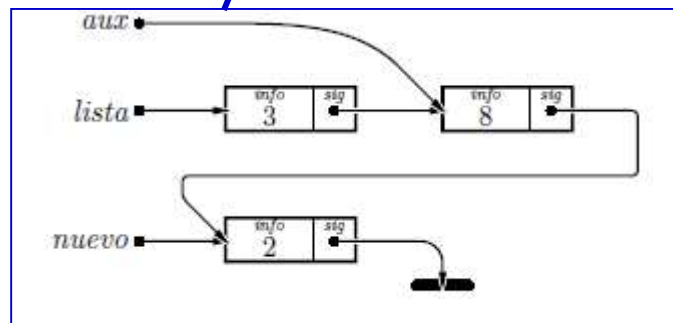
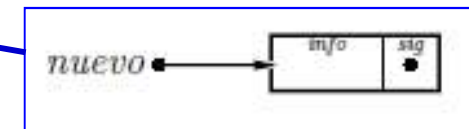
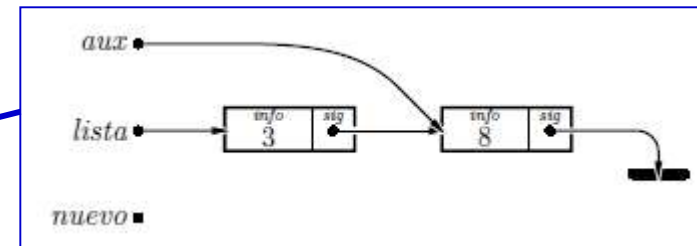


Listas Enlazadas

AGREGAR UN ELEMENTO AL FINAL

```
int main() {
    Nodo * lista = NULL, * aux, * nuevo;

    ...
    aux = lista;
    while (aux->sig != NULL)
        aux = aux->sig;
    nuevo = new Nodo;
    nuevo->info = 2;
    nuevo->sig = NULL;
    aux->sig = nuevo;
}
```



Listas Enlazadas

OTRA FORMA DE AGREGAR UN ELEMENTO AL FINAL

```
Nodo * lista = NULL, * aux, * nuevo;

...
for (aux = lista; aux->sig != NULL; aux = aux->sig);
nuevo = new Nodo;
nuevo->info = 2;
nuevo->sig = NULL;
aux->sig = nuevo;

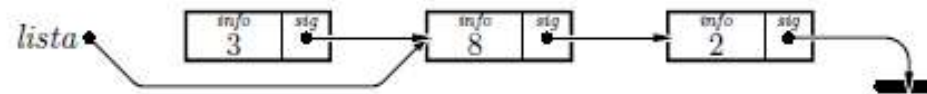
return 0;
```

Bucle *for* sin sentencia en su bloque. Sólo se limita a desplazar el puntero ***aux*** hasta que apunte al último elemento de la lista.

Listas Enlazadas

BORRAR EL PRIMER ELEMENTO

```
1 int main(void)
2 {
3     Nodo * lista = NULL, * aux, * nuevo;
4
5     ...
6     lista = lista->sig; // ¡Mal! Se pierde la referencia a la cabeza original de la lista.
7
8     return 0;
9 }
```



Fugas de memoria: no se puede liberar un bloque de memoria.

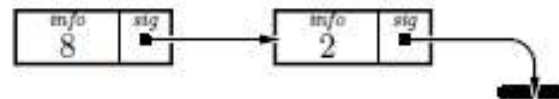
Recolector de basura (Garbage Collector): mecanismo encargado de detectar porciones de memoria que no están en uso y liberarlas → **C no lo implementa.**

Listas Enlazadas

BORRAR EL PRIMER ELEMENTO

```
int main(){  
    Nodo * lista = NULL, * aux, * nuevo;  
  
    ...  
    delete lista;  
    lista = lista->sig; // ;Mal! lista no apunta a una zona de memoria válida  
  
    return 0;  
}
```

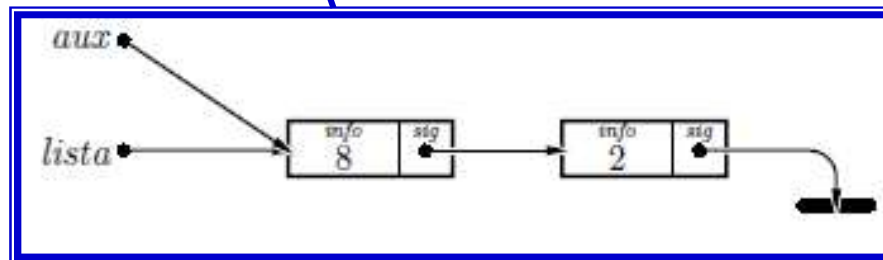
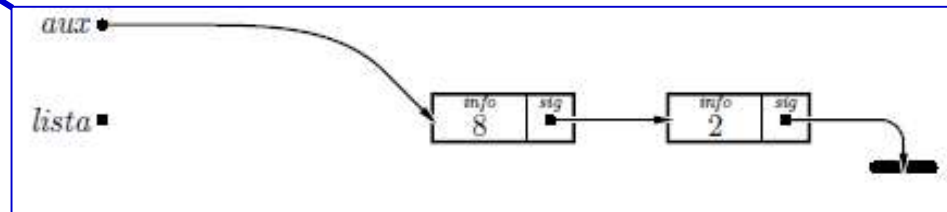
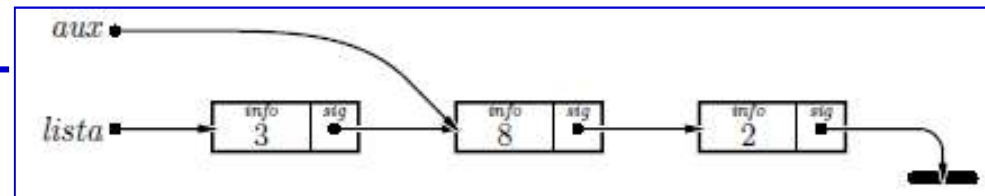
lista ■



Listas Enlazadas

BORRAR EL PRIMER ELEMENTO

```
int main() {  
    Nodo * lista = NULL, * aux, * nuevo;  
  
    ...  
    aux = lista->sig; ←  
    delete lista; ←  
    lista = aux; ←  
    ...  
    return 0;  
}
```



Eliminación exitosa
del 1er elemento
(nodo)

Listas Enlazadas

```
1  /* Operación y mantenimiento de una lista */
2  #include <stdlib.h>
3  #include <iostream>
4  using namespace std;
5
6
7  /* estructura autoreferenciada */
8  struct NodoLista {
9      char info;
10     NodoLista *sig;
11 };
12
13 typedef NodoLista * NodoListaPtr;
14
15 void insertar( NodoListaPtr *sPtr, char valor );
16 char borrar( NodoListaPtr *sPtr, char valor );
17 int vacia( NodoListaPtr sPtr );
18 void printLista( NodoListaPtr actual );
19 void menu( void );
```

Este programa permite:

- 1- Insertar un elemento en la lista (en forma ordenada)
- 2- Borrar un elemento
- 3- Mostrar la lista

Puntero para acceder a la lista

Punteros al puntero que accede a la lista

```

21 int main() {
22     NodoListaPtr startPtr = NULL;
23     int opcion;
24     char item;
25
26     menu();
27     cout << "? ";
28     cin >> opcion;
29
30     while ( opcion != 3 ) {
31         switch ( opcion ) {
32             case 1:
33                 cout << "Ingrese un caracter: ";
34                 cin >> item;
35                 insertar( &startPtr, item );
36                 printLista( startPtr );
37                 break;
38
39             case 2:
40                 if ( !vacía( startPtr ) ) {
41                     cout << "Ingrese caracter a ser borrado: ";
42                     cin >> item;
43
44                     if ( borrar( &startPtr, item ) ) {
45                         cout << item << " borrado.\n";
46                         printLista( startPtr );
47                     }
48                     else
49                         cout << item << " no encontrado.\n\n";
50                 }
51                 else
52                     cout << "Lista vacía.\n\n";
53                 break;

```

Se inicializa la lista como vacía

cout << "Ingrese su eleccion:\n";
 cout << " 1 para insertar un elemento en la lista.\n";
 cout << " 2 para borrar un elemento de la lista.\n";
 cout << " 3 para terminar.\n";

```

55         default:
56             cout << "Opción inválida.\n\n";
57             menu();
58             break;
59     } /* end switch */
60
61     cout << "? ";
62     cin >> opcion;
63 } /* end while */
64
65     cout << "Fin del programa.\n";
66     return 0;
67 } /* end main */
68
69 void menu( void ) {
70     cout << "Ingrese su eleccion:\n";
71     cout << "    1 para insertar un elemento en la lista.\n";
72     cout << "    2 para borrar un elemento de la lista.\n";
73     cout << "    3 para terminar.\n";
74 }
75

```

```

76  /* Inserta un nuevo nodo en la lista en forma ordenada */
77  void insertar( NodoListaPtr *sPtr, char valor ) {
78      NodoListaPtr nuevo;
79      NodoListaPtr anterior;
80      NodoListaPtr actual;
81
82      nuevo = new NodoLista;
83
84      if ( nuevo != NULL ) { /* si hay espacio disponible */
85          nuevo->info = valor;
86          nuevo->sig = NULL;
87
88          anterior = NULL;
89          actual = *sPtr;
90
91          /* busca la posición correcta en la lista */
92          while ( actual != NULL && valor > actual->info ) {
93              anterior = actual;      /* avanza al ... */
94              actual = actual->sig;    /* ... próximo nodo */
95          }
96
97          /* inserta nuevo al comienzo de la lista */
98          if ( anterior == NULL ) {
99              nuevo->sig = *sPtr;
100             *sPtr = nuevo;
101         }
102         else { /* inserta nuevo entre anterior y actual */
103             anterior->sig = nuevo;
104             nuevo->sig = actual;
105         }
106     }
107     else
108         cout << valor << " no insertado. No hay memoria disponible.\n";
109 }
110

```

Se crea un nuevo nodo (se pide espacio y se inicializa)

Ahora, la lista comienza en el nuevo nodo insertado

```

111  /* Borra un elemento de la lista */
112  char borrar( NodoListaPtr *sPtr, char valor ) {
113      NodoListaPtr anterior;
114      NodoListaPtr actual;
115      NodoListaPtr tempPtr;
116
117      /* borra el primer nodo */
118      if ( valor == ( *sPtr )->info ) {
119          tempPtr = *sPtr;
120          *sPtr = ( *sPtr )->sig; /* desengancha el 1er nodo */
121          delete tempPtr;        /* libera memoria */
122          return valor;
123      } /* end if */
124      else {
125          anterior = *sPtr;
126          actual = ( *sPtr )->sig;
127
128          /* ciclo para encontrar la posición correcta en la lista */
129          while ( actual != NULL && actual->info != valor ) {
130              anterior = actual; /* avanza al ... */
131              actual = actual->sig; /* ... próximo nodo */
132          }
133
134          /* borra nodo en actual */
135          if ( actual != NULL ) {
136              tempPtr = actual;
137              anterior->sig = actual->sig;
138              delete tempPtr;
139              return valor;
140          }
141      } /* end else */
142      return '\0';
143  } /* end funcion borrar */
144

```



```

145  /* Devuelve 1 si la lista está vacía, 0 en otro caso */
146  int vacia( NodoListaPtr sPtr ) {
147      return sPtr == NULL;
148  }
149
150  /* Imprime la lista */
151  void printLista( NodoListaPtr actual ) {
152      /* si la lista está vacía */
153      if ( actual == NULL )
154          cout << "La lista está vacía.\n\n" ;
155      else {
156          cout << "La lista es:\n";
157
158          /* mientras no sea el fin de la lista */
159          while ( actual != NULL ) {
160              cout << actual->info << " --> ";
161              actual = actual->sig;
162          }
163
164          cout << "NULL\n\n";
165      } /* end else */
166  }
167
168

```

Sólo analiza si el puntero a la lista es nulo

Opción: 1) insertar un elemento al final

```
Ingrese su eleccion:
    1 para insertar un elemento en la lista.
    2 para borrar un elemento de la lista.
    3 para terminar.
? 1
Ingrese un caracter: A
La lista es:
A --> NULL

? 1
Ingrese un caracter: H
La lista es:
A --> H --> NULL

? 1
Ingrese un caracter: F
La lista es:
A --> F --> H --> NULL

? 1
Ingrese un caracter: B
La lista es:
A --> B --> F --> H --> NULL

?
```



```
La lista es:  
A --> B --> F --> H --> NULL  
  
? 2  
Ingrese caracter a ser borrado: F  
F borrado.  
La lista es:  
A --> B --> H --> NULL
```

Opción: 3) borrar un elemento

```
? 2  
Ingrese caracter a ser borrado: B  
B borrado.  
La lista es:  
A --> H --> NULL  
  
? 2  
Ingrese caracter a ser borrado: H  
H borrado.  
La lista es:  
A --> NULL
```

```
? 2  
Ingrese caracter a ser borrado: A  
A borrado.  
La lista esta vacia.
```

```
? 2  
Lista vacia.
```

```
? 7  
Opcion invalida.
```

```
Ingrese su eleccion:  
1 para insertar un elemento en la lista.  
2 para borrar un elemento de la lista.  
3 para terminar.
```

```
? 3  
Fin del programa.
```

```
<< El programa ha finalizado: codigo de salida: 0 >>  
<< Presione enter para cerrar esta ventana >>_
```

Otras funciones

1. Insertar al final
2. Imprimir la lista en modo inverso (del último al primero)
3. Obtener el menor elemento de una lista
4. Ordenar una lista por uno de sus campos
5. Intercalar 2 listas
6. Borrar el primero, borrar la lista (de inicio a fin, o en sentido inverso)

Ejemplo con más funciones

```
1  /* Operación y mantenimiento de una lista */
2  #include <stdlib.h>
3  #include <iostream>
4  using namespace std;
5
6
7  /* estructura autoreferenciada */
8  struct NodoLista {
9      char info;
10     struct NodoLista *sig;
11 };
12
13 typedef NodoLista * NodoListaPtr;
14
15 void insertar( NodoListaPtr *sPtr, char valor );
16 char borrar( NodoListaPtr *sPtr, char valor );
17 int vacia( NodoListaPtr sPtr );
18 void printLista( NodoListaPtr actual );
19 void menu( void );
20 void insertf (NodoListaPtr *sPtr, char valor );
21 void printListaR( NodoListaPtr actual );
```

1. y 2.
pueden ser
recursivas!

```

while ( opcion != 5 ) {
    switch ( opcion ) {
        case 1:
            cout << "Ingreso un caracter: ";
            cin >> item;
            insertar( &startPtr, item );
            printLista( startPtr );
            break;

        case 2:
            if ( !vacía( startPtr ) ) {
                cout << "Ingreso caracter a ser borrado: ";
                cin >> item;

                if ( borrar( &startPtr, item ) ) {
                    cout << item << " borrado.\n";
                    printLista( startPtr );
                }
                else
                    cout << item << " no encontrado.\n\n";
            }
            else
                cout << "Lista vacía.\n\n";
            break;

        case 3:
            cout << "Ingreso un caracter: ";
            cin >> item;
            insertar( &startPtr, item );
            printLista( startPtr );
            break;

        case 4:
            printListaR( startPtr );
            cout << endl << endl;
            break;
    }
}

```

```

/* Inserta un nuevo nodo al final de la lista */
void insertf( NodoListaPtr *sPtr, char valor ) {
    NodoListaPtr nuevo;

    if (*sPtr == NULL) {
        nuevo = new NodoLista;
        if ( nuevo != NULL ) {      /* si hay espacio disponible */
            nuevo->info = valor;
            nuevo->sig = NULL;
            *sPtr = nuevo;}
        else cout << "No hay espacio";
    }
    else insertf (&(*sPtr)-> sig,valor);
}

```

```

1 para insertar un elemento en la lista.
2 para borrar un elemento de la lista.
3 para insertar un elemento al final de la lista.
4 para terminar.
? 1
Ingrese un caracter: a
La lista es:
a --> NULL

? 1
Ingrese un caracter: g
La lista es:
a --> g --> NULL

? 1
Ingrese un caracter: d
La lista es:
a --> d --> g --> NULL

? 3
Ingrese un caracter: b
La lista es:
a --> d --> g --> b --> NULL

?

```

```

/* Imprime la lista en orden inverso */
void printListaR( NodoListaPtr actual ) {

    /* si la lista está vacía */
    if ( actual == NULL )
        cout << "NULL";
    else {
        printListaR (actual->sig);
        cout << " --> " << actual->info;
    }
}

```

```

Ingrese su eleccion:
1 para insertar un elemento en la lista.
2 para borrar un elemento de la lista.
3 para insertar un elemento al final de la lista.
4 para imprimir en orden inverso.
5 para terminar.
? 1
Ingrese un caracter: A
La lista es:
A --> NULL

? 1
Ingrese un caracter: H
La lista es:
A --> H --> NULL

? 1
Ingrese un caracter: F
La lista es:
A --> F --> H --> NULL

? 1
Ingrese un caracter: B
La lista es:
A --> B --> F --> H --> NULL

? 4
NULL --> H --> F --> B --> A


? 5
Fin del programa.

```

Menor elemento de una lista

```
PtrNode menorLista( PtrNode P ){  
    // la lista no es vacía  
    PtrNode aux= P, menorptr= P;  
    char menor= aux->info;  
  
    while (aux != NULL){  
        if (aux->info < menor){  
            menor= aux->info;  
            menorptr= aux;  
        }  
        aux=aux->sig;  
    }  
    return menorptr;  
}
```

Ordenar una lista



```
void ordenarlista(PtrNode L1){  
    if (L1 == NULL || L1->sig == NULL)  
        return;  
    else {  
        PtrNode aux= menorLista( L1->sig );  
        if (L1->info > aux->info )  
            intercambiar( &(L1->info) , &(aux->info) );  
        ordenarlista(L1->sig);  
    }  
}
```

```
void intercambiar (char * a, char * b) {  
    char aux= *a;  
    *a = *b;  
    *b =aux; }  
}
```


Intercalar dos listas

```
void mergeLists(PtrNodo L1, PtrNodo L2, PtrNodo * L3){
    PtrNodo aux1=L1, aux2=L2;

    while (aux1!=NULL && aux2!=NULL){
        if (aux1->info < aux2->info){
            insfin(L3, aux1->info);
            aux1= aux1->sig;
        }
        else {
            insfin(L3, aux2->info);
            aux2= aux2->sig;
        }
    }
    while (aux1!=NULL){
        insfin(L3, aux1->info);
        aux1= aux1->sig;
    }
    while (aux2!=NULL){
        insfin(L3, aux2->info);
        aux2= aux2->sig;
    }
}
```

Opciones de borrado

```
void borrarpri( PtrNode *sPtr ){  
    /* funcion que borra el primer nodo de una lista */  
    PtrNode temp;  
  
    if (*sPtr != NULL) {  
        temp = *sPtr;  
        *sPtr = (*sPtr)->sig;  
        delete temp;  
    }  
}  
  
void liberarlista ( PtrNode *sPtr ){  
    /* funcion iterativa para borrar una lista completa */  
    while (*sPtr != NULL)  
        borrarpri(sPtr);  
}  
  
void liberarlistaR ( PtrNode *sPtr ){  
    /* funcion recursiva para borrar una lista completa */  
    if (*sPtr != NULL) {  
        borrarpri(sPtr);  
        liberarlistaR(sPtr);  
    }  
}
```

**Borrar el primer
nodo**

**Borrar
la
lista**

Otras listas

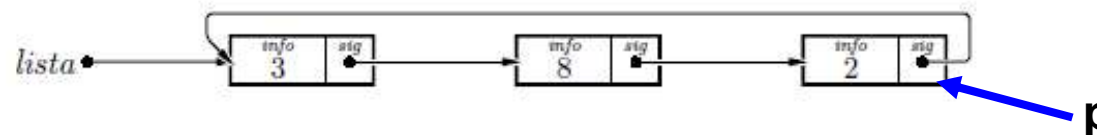
■ Listas circulares

- Lista enlazada simple en la que el último nodo referencia al primero.
- No hay un principio y fin definido.
- Ningún nodo está enlazado a NULL.
- El puntero se usa para reconocer un ingreso a la lista.



- Como el último nodo tiene referencia al primero, se puede mantener un puntero (p) apuntando al último nodo y desde allí acceder al primero a través de $p \rightarrow \text{sig}$.

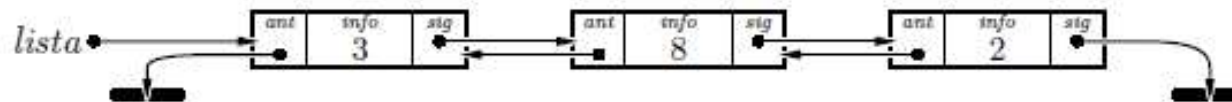
Así en una lista circular se tiene referenciado el último nodo e indirectamente el primero.



Otras listas

- Listas doblemente enlazadas

```
struct unnodo {  
    int info;  
    unnodo *ant;  
    unnodo *sig;  
};
```



Cada nodo tiene un puntero al siguiente y otro al anterior nodo

Se pueden mantener punteros al primero y al último nodo

Utiliza más memoria pero tiene las siguiente ventajas:

- La lista puede recorrerse en ambas direcciones
- Las operaciones insertar y eliminar utilizan menor cantidad de instrucciones

LEER

Capítulo 13

“Memoria dinámica. Punteros”

**Libro: “Fundamentos de Programación con el
Lenguaje de Programación C++” – Benjumea y
Roldán, pág. 163**

“Punteros y datos dinámicos”, pág. 895

“Listas Enlazadas”, pág. 964

Libro: “Fundamentos de la programación” – Yáñez,