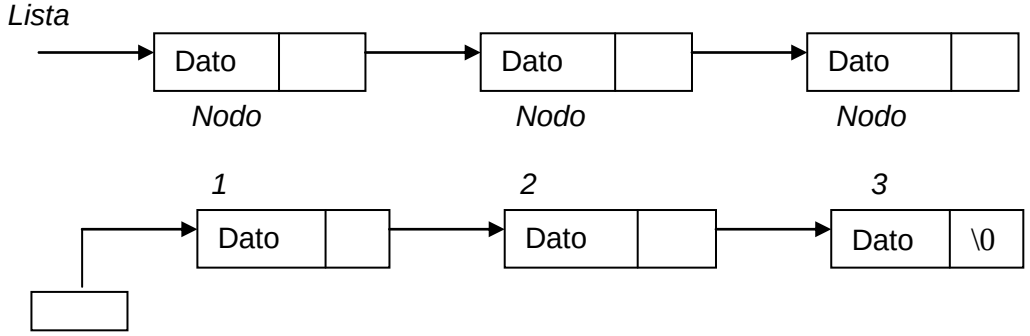
UNIVERSIDAD DE EL SALVADOR.
FACULTAD DE INGENIERIA Y ARQUITECTURA.
ESCUELA DE SISTEMAS INFORMATICOS.
ESTRUCTURAS DE DATOS

Listas eslabonadas

Definición:

Colección de elementos o nodos en que cada uno esta formado de dos partes: La primera hace referencia a los datos (Data o Información) y la segunda re refiere al enlace (Puntero al siguiente elemento de la lista) Algunos sinónimos con los que se conocen las listas son los siguientes: listas enlazadas, listas eslabonadas.

Gráficamente:



Puntero externo

Características:

1- Debe de haber un puntero enlace externo que señale el inicio de la lista (List).
2- No pueden accesarse aleatoriamente a los nodos de una lista eslabonada.
3- El último nodo tiene como enlace un valor nulo.
4- La lista que no contiene nodo se denomina lista vacía o lista nula (List =Null).

Las operaciones que podemos realizar sobre una lista enlazada son las siguientes:

- **Recorrido.** Esta operación consiste en visitar cada uno de los nodos que forman la lista . Para recorrer todos los nodos de la lista, se comienza con el primero, se toma el valor del campo liga para avanzar al segundo nodo, el campo liga de este nodo nos dará la dirección del tercer nodo, y así sucesivamente.

- **Inserción.** Esta operación consiste en agregar un nuevo nodo a la lista. Para esta operación se pueden considerar tres casos:
  - Insertar un nodo al inicio.
  - Insertar un nodo antes o después de cierto nodo.
  - Insertar un nodo al final.

- **Borrado.** La operación de borrado consiste en quitar un nodo de la lista, redefiniendo las ligas que correspondan. Se pueden presentar cuatro casos:
  - Eliminar el primer nodo.
  - Eliminar el último nodo.

- Eliminar un nodo con cierta información.
- Eliminar el nodo anterior o posterior al nodo cierta con información.

- **Búsqueda.** Esta operación consiste en visitar cada uno de los nodos, tomando al campo liga como puntero al siguiente nodo a visitar.

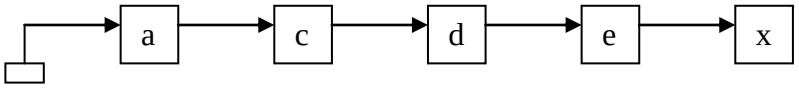
Criterios para definir tipos de lista.

Orden de los elementos:
- Ascendentes
- Descendentes

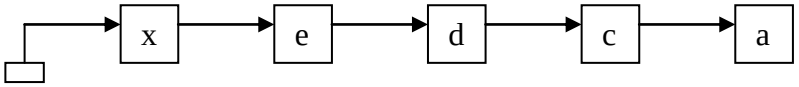
Acceso a la lista o direccionamiento:
- Unidireccional
- Bidireccional

Representación de algoritmos listas:

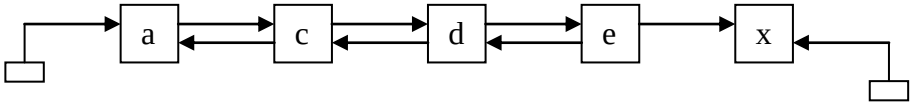
1- Lista ordenada ascendente-unidireccional



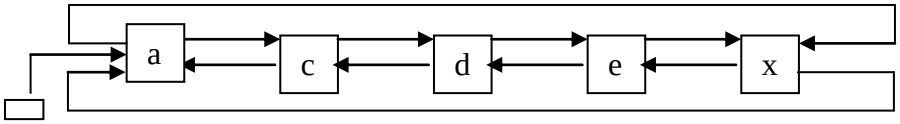
2- Lista ordenada descendente- unidireccional



3- Combinación 1 y 2



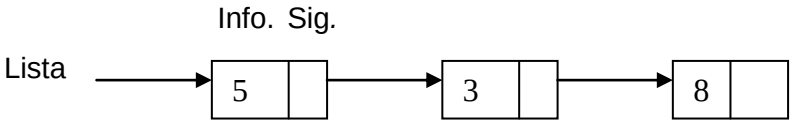
4- Listas circular



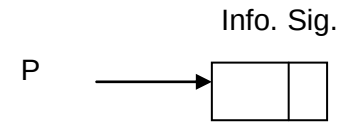
Notación a usar en algoritmos.

P = Puntero a un nodo
NODO (P)= Hacer referencia al nodo al que apunta P.
INFO (P)= Hace referencia a la información del nodo.
NEXT (P)= Siguiente dirección (puntero)si next (p) no es NULL.
INFO (NEXT(P))= Información de nodo que sigue a nodo (p) en la lista.

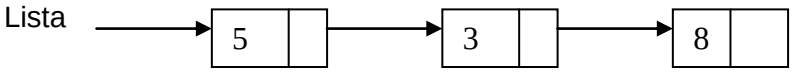
Inserción y remoción de nodo de una lista.

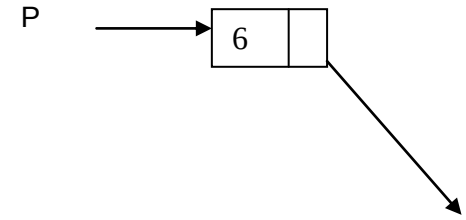


Agregar 6 al frente de la Lista



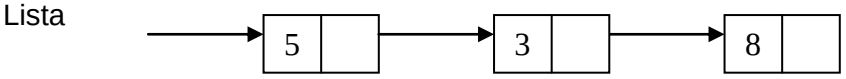
P= getnode( )



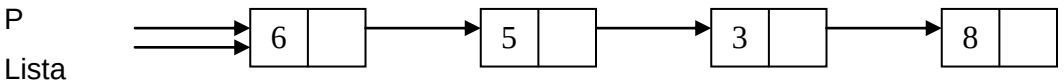


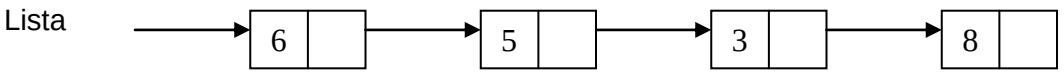
= > info (P) = 6

=> next (P) =Lista



Lista =P

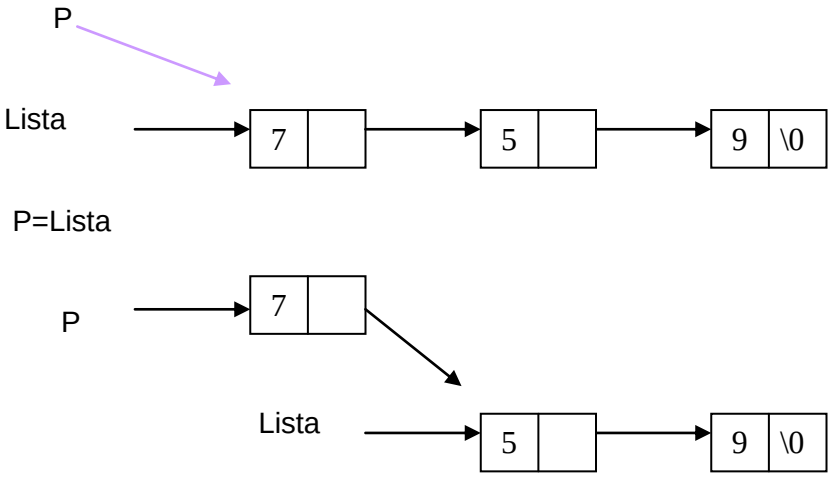




Algoritmo:

```
P= getnode ();
info(P)= 6;
next (P)= Lista;
Lista = P;
```

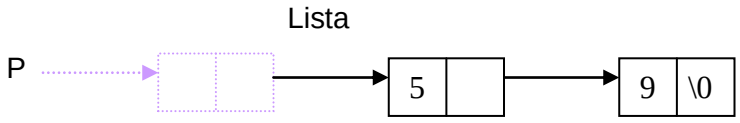
Remover el primer nodo de un lista.



Lista =next(P)

X=info(P)=7

Freenode(P)



Algoritmo:

```
P= Lista;
Lista = next (P);
X= info (P);
Freenode (P);
```

Operaciones Getnode y freenode.

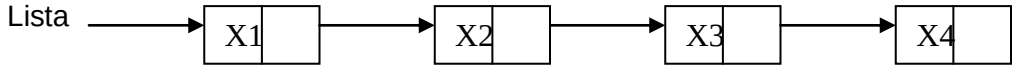
```
P= getnode ();
Se implementa:
        If (avail== NULL){
                Printf ("overflow");
                Exit (1);
                 }
        P= avail;
        avail= next(avail);

Freenode(P);      /* Agregar un nuevo nodo */
```

Se implementa:

next(P)=avail;
avail=P;

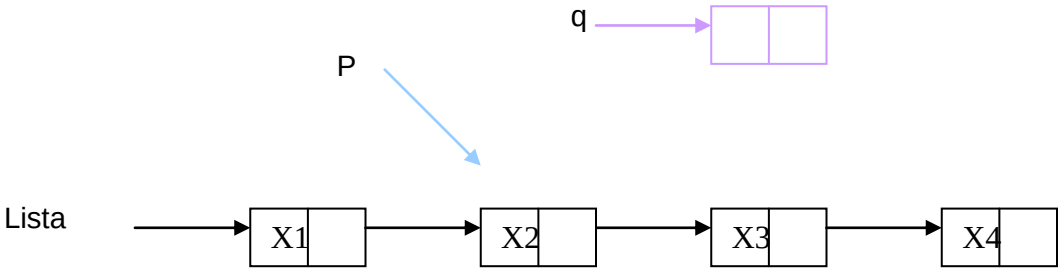
La lista eslabonada como estructura de datos.

Operación de Inserción de nodos



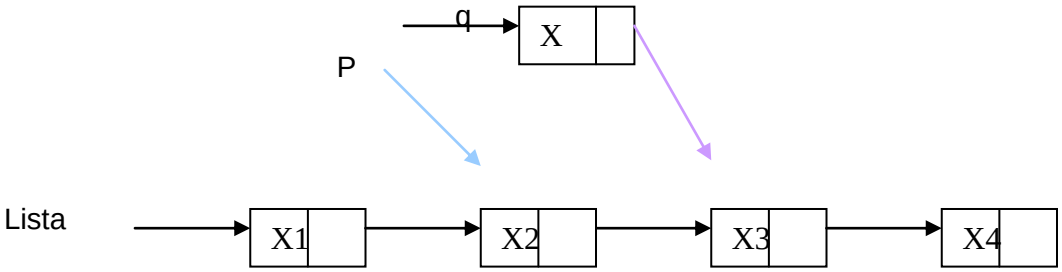
Nuevo Nodo con Info=X

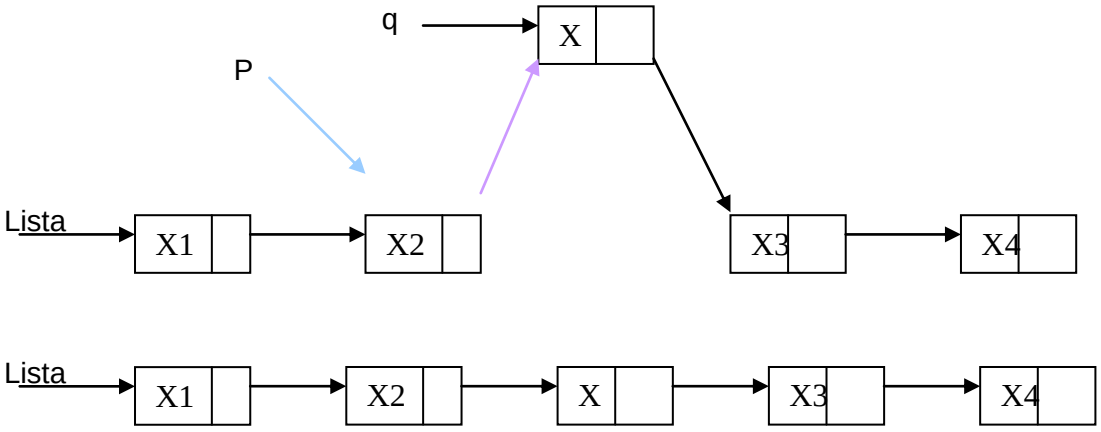
Insafter (P,X) (Agrega un nuevo Nodo)



Algoritmo:

Insafter (p, x) /* Inserta un nuevo nodo*/
q= getnodo ();
Info (q)= x;
Next(q)= Next (p);
next (p)= q;

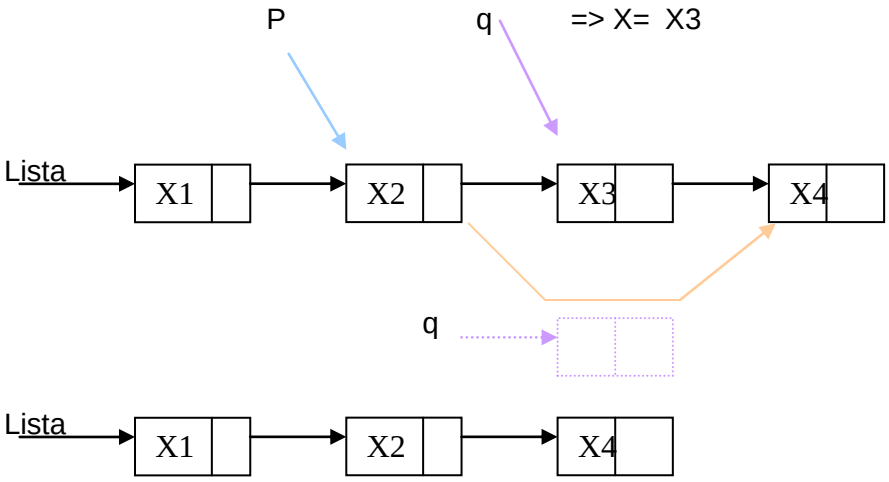




Operación de eliminación de nodos

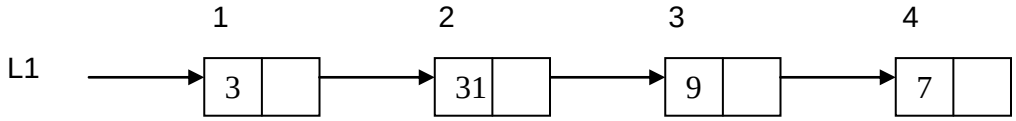
Algoritmo:
Delafter (p, x)

```
        q= next ();
        x=  info (q);
        next (p)= next (q);
        freenode (q);
```

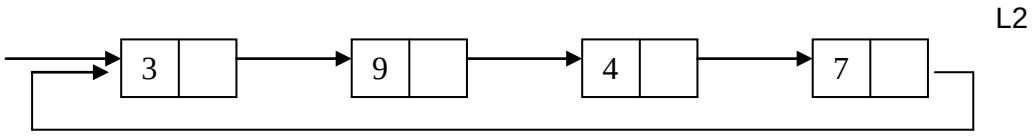


Implementación de lista de colas de prioridad.

Nodo de encabezado:

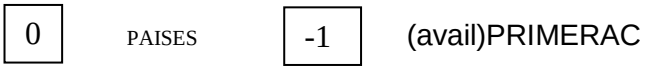


1 - Nodo de Encabezado : Contiene el numero de nodos siguientes que utiliza la lista.

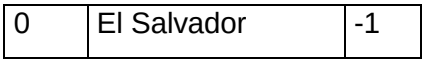


Implementación de Listas Ascendentes y Unidireccionales

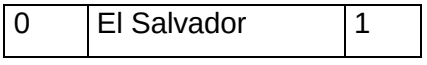
Supongamos un arreglo de n elementos los cuales pueden contener variables de tipo carácter. Podemos asegurar que al inicio ese arreglo es una lista vacía de nodos disponibles la cual es accesada por la variable primerac.



Adicionar (países, 'El Salvador ')



Adicionar (países, 'Guatemala ')



1	Guatemala	-1

2 (avail)PRIMERAC 0 PAISES

Adicionar (países, ‘Honduras ‘)

0	El Salvador	1
1	Guatemala	2
2	Honduras	-1

3 (avail)PRIMERAC 0 PAISES

Adicionar (Alumnos, ‘ Juan Pérez ‘)

0	El Salvador	1
1	Guatemala	2
2	Honduras	-1
3	Juan Pérez	-1

4 (avail)PRIMERAC 0 PAISES 3 ALUMNOS

Adicionar (Alumnos, ‘Ana Ramírez ‘)

0	El Salvador	1
1	Guatemala	2
2	Honduras	-1
3	Juan Pérez	4
4	Ana Ramírez	-1

5 (avail)PRIMERAC 0 PAISES

3 ALUMNOS

Adicionar (Herramientas, ‘ Tenaza ‘)

0	El Salvador	1
1	Guatemala	2
2	Honduras	-1

3	Juan Pérez	5
4	Ana Ramírez	-1
5	Tenaza	-1

6	(avail)PRIMERAC	0	PAISES
3	ALUMNOS	5	HERRAMIENTAS

Eliminar (Países, 'Guatemala')

0	El Salvador	1
~~1~~	~~Guatemala~~	~~2~~
2	Honduras	-1
3	Juan Pérez	5
4	Ana Ramírez	-1
5	Tenaza	-1

1	(avail)PRIMERAC	0	PAISES
3	ALUMNOS	5	HERRAMIENTAS

Adicionar (Herramientas, 'Destornillador')

0	El Salvador	2
1	Destornillador	-1
2	Honduras	-1
3	Juan Pérez	4
4	Ana Ramírez	-1
5	Tenaza	1

6	(avail)PRIMERAC	0	

HERRAMIENTAS

PAISES

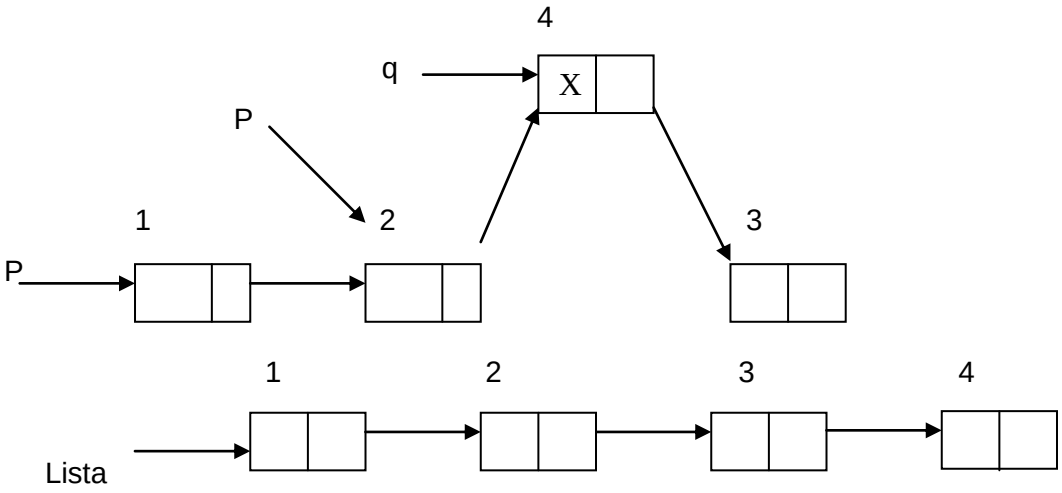
3 ALUMNOS 1

Operaciones GETNODE Y FREENODE

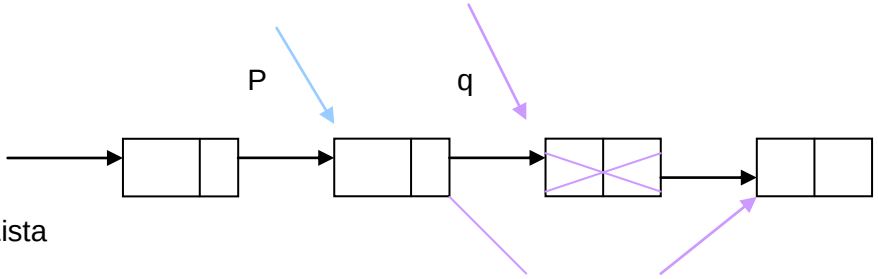
```
int getnode ( void){
        int p;
        if ( avail== -1){
                printf (“overflow \n”);
                exit (1);
        }
        p= avail ;
        avail= node [avail].next;
return   (p);
}

void insafter (int p, int x ){
        int q ;
        if (p== -1){
                prinf (“fallo la inserción \n”)
                return ();
        }
        q= getnode ();
        node[q].info =x;
        node[q].next =node[p].next;
        node[p].next =q;
        return ;
}


void delafter (int p, int *px)
{
```

```
        int q;
        if ((p==-1)|| node[p].next==-1){
                printf ("fallo eliminación \n")
                return ;
        }
        q= node [p].next;
        *px= node [q].info;
        node[p].next= node [q].next;
        freenode (q);
return;
} /* fin de desafter*/
```

INSAFTER:

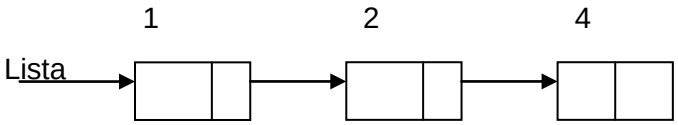


DELAFTER



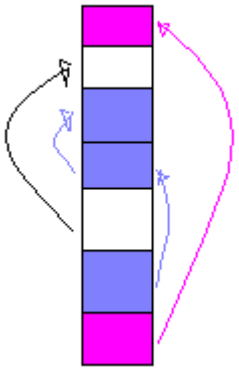
Lista

Px= node [q].info



Una Lista puede Contener varias listas, estas estarán Enlazadas entre si por los respectivos punteros, ya que cada nodo apuntaran al siguiente objeto que le corresponda en la lista .

En una forma gráfica se podría definir así ⟶



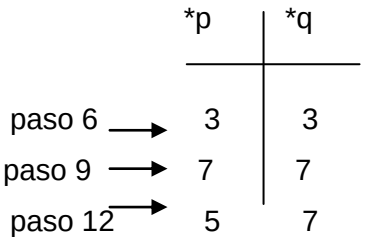
Asignación y definición de variables dinámicas.

x= objeto variable
&x=puntero a x
P= puntero
*p= objeto al que apunta P

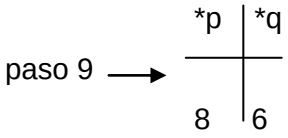
```
extern char *malloc ();
int *pi;
float *pr;
pi= (int  *)malloc (sizeof (int));
pr =(float *) malloc (sizeof (float));
```

1. int *p,*q;
2. int x;
3. p= (int *)malloc (sizeof (int));
4. *p=3;
5. q = p;

6. printf (“%d %d\n ”,*p,*q);
7. x=7;
8. *q= x;
9. printf (“%d%d”, * p**,** ***q);
10. p= (int *)malloc (sizeof (int));
11. *p=5;
12. printf (“%d, %d”,*p,*q);



1. p= (int *)malloc (sizeof (int));
2. *p=5;
3. q=(int *)malloc (sizeof(int));
4. *q=8;
5. free(p);
6. p=q;
7. q=(int *)malloc (sizeof(int));
8. *q=6;
9. printf (“%d%d\n”,*p,*q);



Observe que si se llama malloc dos veces sucesivas y se asigna su valor a la misma variable como en el ejemplo anterior se pierde la primera copia de *p dado que su dirección no se guardo.

El espacio asignado para variables dinámicas solo puede accesarse mediante un puntero. A menos que su guarde el puntero a la primera variable dentro de otro puntero la variable se perderá. De hecho su almacenamiento ni siquiera se puede liberar dado que no hay modo de hacer referencia a ella en la llamada a free() esto es ejemplo de almacenamiento asignado al que no se puede tener referencia.

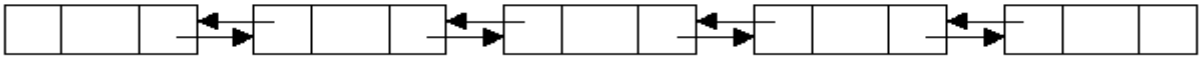
Listas vinculadas usando variables dinámicas.

```
Struct node {
        Int info;
Struct node, *next;
}
typedef struct node *nodeptr;
nodeptr p;
p= getnode ()

nodeptr  getnode (void){
nodeptr p;
        p= (nodeptr)malloc (sizeof(struct node));
        return ( p);
}
freenode (p)

void  freenode(nodeptr p){
        free (p);
}

void insafter (nodeptr p){
nodeptr q;
if (p== NULL){
        printf (“fallo inserción \n”);
        exit (1)
}
q= getnode ();
q->info= x;
q->next= q;
}/* fin insafter */

void delafter (nodeptr p, int *px){
        nodeprt q;
        if ( (p==NULL)||(p-> next== NULL)){
                printf (“”fallo eliminación \n);
                exit (1);
        }
        q= p->next;
        *px= q-Zinfo;
        p->next= q->next;
        freenode (q);
}/* fin de delafter */
```

Listas doblemente enlazadas

*Introducció*n.

En algunas aplicaciones podemos desear recorrer la lista hacia adelante y hacia atrás, o dado un elemento, podemos desear conocer rápidamente los elementos anterior y siguiente. En tales situaciones podríamos desear darle a cada celda sobre una lista un puntero a las celdas siguiente y anterior en la lista tal y como se muestra en la figura.



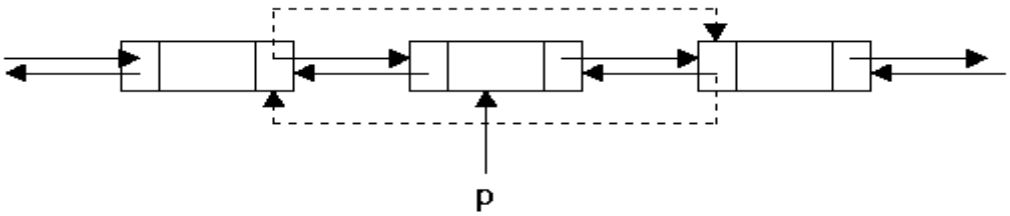
El único precio que pagamos por estas características es la presencia de un puntero adicional en cada celda y consecuentemente procedimientos más largos para algunas de las operaciones básicas de listas. Si usamos punteros, podemos declarar celdas que consisten en un elemento y dos punteros a través de:

```
typedef int tElemento;

typedef struct celda {
   tElemento elemento;
   struct celda *siguiente,*anterior;
} tipocelda;

typedef tipocelda *tPosicion;
typedef tipocelda *tLista;
```

Un procedimiento para borrar un elemento en la posición p en una lista doblemente enlazada es:

```
void borrar (posicion p)
{
   if (p->anterior != NULL)
         p->anterior->siguiente = p->siguiente;
   if (p->siguiente != NULL)
         p->siguiente->anterior = p->anterior;
   free(p);
}
```

El procedimiento anterior se expresa de forma gráfica en como muestra la figura anterior. Donde los trazos continuos denotan la situación inicial y los punteados la final. El ejemplo visto se ajusta a la eliminación de un elemento o celda de una lista situada en medio de la misma.



Implementación de listas doblemente Enlazadas

Respecto a la forma en que trabajarán las funciones de la implementación que proponemos hay que hacer constar los siguientes puntos:

La función *posición* retorna la dirección de memoria del último nodo de la lista, si es que no se ha encontrado, porque de ser así retorna la posición a donde lo encuentra. Además es de hacer notar que la función busca hasta el fin la lista o hasta encontrar el elemento.

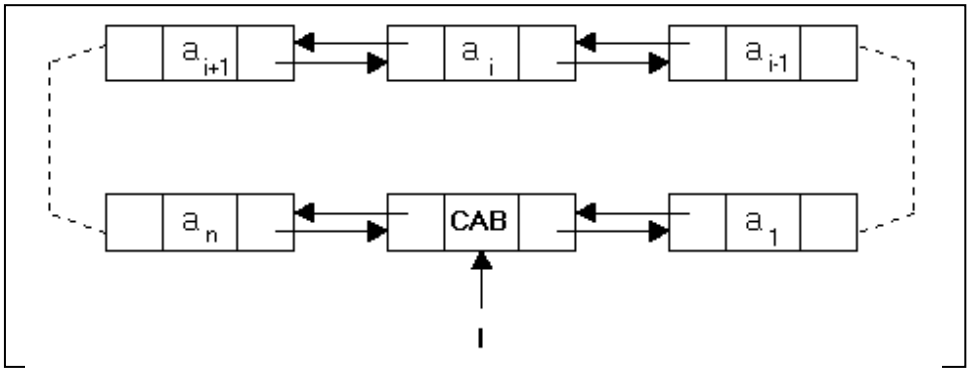
La función de *inserción* adiciona el nuevo nodo al final de la lista, pero si es encontrado se inserta justo después, creándose un duplicado. La función considera los casos de insertar al principio, en medio y al final de la lista.

La función de *Borrar* elimina el nodo apuntado por el parámetro *p, encontrado por la función *posición.* Igualmente considera la eliminación al inicio, en medio y al final de la lista.

Colas doblemente enlazadas circulares.

Para obviar los problemas derivados de los elementos extremos (primero y último) es práctica común hacer que la cabecera de la lista doblemente enlazada sea una celda que efectivamente complete el círculo, es decir, el anterior a la celda de cabecera sea la última celda de la lista y la siguiente la primera. De esta manera no necesitamos chequear para NULL al final o inicio de la lista.

Por consiguiente, podemos realizar una implementación de listas doblemente enlazadas con cabecera tal que tenga una estructura circular en el sentido de que dado un nodo y por medio de los punteros *siguiente* podemos volver hasta él como se puede observar en la figura (de forma análoga para *anterior*).



Es importante notar que aunque la estructura física de la lista puede hacer pensar que mediante la operación *siguiente* podemos alcanzar de nuevo un nodo de la lista, la estructura lógica es la de una lista y por lo tanto habrá una posición *primero* y una posición *fin* de forma que al aplicar una operación *anterior* o *siguiente* respectivamente sobre estas posiciones el resultado será un error.

Respecto a la forma en que trabajarán las funciones de la implementación que proponemos hay que hacer constar los siguientes puntos:

La función de creación debe alojar memoria para la cabecera y hacer que los punteros *siguiente* y *anterior* apunten a ella, devolviendo un puntero a dicha cabecera.

La función *primero(l)* devolverá un puntero al nodo siguiente a la cabecera.
La función *fin(l)* devolverá un puntero al nodo cabecera.

En la implementación final optaremos por pasar un puntero a la posición para el borrado de forma que la posición usada quede apuntando al elemento siguiente que se va a borrar al igual que ocurría en el caso de las listas simples.

La inserción se hará a la izquierda del nodo apuntado por la posición ofrecida a la función insertar. Esto implica que al contrario que en las listas simples, al insertar un nodo, el puntero utilizado sigue apuntando al mismo elemento al que apuntaba y no al nuevo elemento insertado.

Operaciones de listas dobles circulares.

Dentro del tipo abstracto de *listas doblemente enlazadas* podemos proponer las siguientes primitivas:

```
tLista crear ()
void destruir (tLista l)
tPosicion primero (tLista l)
tPosicion fin (tLista l)
void insertar (tElemento x, tPosicion p, tLista l)
void borrar (tPosicion *p, tLista l)
tElemento elemento(tPosicion p, tLista l)
tPosicion siguiente (tPosicion p, tLista l)
tPosicion anterior (tPosicion p, tLista l)
tPosicion posicion (tElemento x, tLista l)
```

Implementación de listas doblemente enlazadas circulares.

Una vez aclaradas las posibles ambigüedades y dudas que se pueden plantear, la implementación de las listas doblemente enlazadas quedaría como sigue:

```
typedef struct celda {
   tElemento elemento;
   struct celda *siguiente,*anterior;
} tipocelda;

typedef tipocelda *tPosicion;
typedef tipocelda *tLista;

static void error(char *cad) {
   fprintf(stderr, "ERROR: %s\n", cad);
   exit(1);
}

tLista Crear() {
   tLista l;
```

```
  l = (tLista)malloc(sizeof(tipocelda));
  if (l == NULL)
        Error("Memoria insuficiente.");
  l->siguiente = l->anterior = l;
  return l;
}

void Destruir (tLista l) {
  tPosicion p;

  for (p=l, l->anterior->siguiente=NULL; l!=NULL; p=l) {
        l = l->siguiente;
        free(p);
  }
}


tPosicion Primero (tLista l) {
  return l->siguiente;
}

tPosicion Fin (tLista l) {
  return l;
}


void Insertar (tElemento x, tPosicion p, tLista l) {
  tPosicion nuevo;

  nuevo = (tPosicion)malloc(sizeof(tipocelda));
  if (nuevo == NULL)
        Error("Memoria insuficiente.");
  nuevo->elemento = x;
  nuevo->siguiente = p;
  nuevo->anterior = p->anterior;
  p->anterior->siguiente = nuevo;
  p->anterior = nuevo;
}


void Borrar (tPosicion *p, tLista l) {
  tPosicion q;
  if (*p == l){
        Error("Posicion fin(l)");
  }
  q = (*p)->siguiente;
  (*p)->anterior->siguiente = q;
  q->anterior = (*p)->anterior;
  free(*p);
  (*p) = q;
}
```

```
tElemento elemento(tPosicion p, tLista l) {
  if (p == l){
        Error("Posicion fin(l)");
  }
  return p->elemento;
}

tPosicion siguiente (tPosicion p, tLista l) {
  if (p == l){
        Error("Posicion fin(l)");
  }
  return p->siguiente;
}


tPosicion anterior( tPosicion p, tLista l) {
  if (p == l->siguiente){
        Error("Posicion primero(l)");
  }
  return p->anterior;
}


tPosicion posicion (tElemento x, tLista l)
{
  tPosicion p;
  int encontrado;

  p = primero(l);
  encontrado = 0;
  while ((p != fin(l)) && (!encontrado))
        if (p->elemento == x)
              encontrado = 1;
        else
              p = p->siguiente;
  return p;
}
```